1-1-1993

On strong reduction in director string calculus

Matthew Edward Kinkenon

University of Nevada, Las Vegas

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

Repository Citation


This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
INFORMATION TO USERS

This manuscript has been reproduced 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.
On strong reduction in director string calculus

Kinkenon, Matthew Edward, M.S.
University of Nevada, Las Vegas, 1993
On Strong Reduction in Director String Calculus

by

Matthew Edward Kinkenon

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

Master of Science
in
Computer Science

Department of Computer Science
University of Nevada, Las Vegas
May 1993
The thesis of Matthew E. Kinkenon for the degree of Master of Science in Computer Science is approved.

Chairperson, Kazem Taghva, Ph.D

Examineing Committee Member, Laxmi Gewali, Ph.D

Examineing Committee Member, John Minor, Ph.D

Graduate Faculty Representative, Ebrahim Salehi, Ph.D

Graduate Dean, Ronald W. Smith, Ph.D

University of Nevada, Las Vegas
May, 1993
Abstract

An algebra is presented which is used in conjunction with a modified version of director string calculus (MDSC). The algebra simplifies the implementation of the MDSC while preserving the applicative structure of the original expression, and capturing strong reduction.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>iii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>v</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>vi</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Overview</td>
<td>1</td>
</tr>
<tr>
<td>1.2 History of Computing</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Functional Languages</td>
<td>3</td>
</tr>
<tr>
<td>2 Compilation</td>
<td>6</td>
</tr>
<tr>
<td>2.1 Overview</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Variables and Functions</td>
<td>6</td>
</tr>
<tr>
<td>2.3 Constructors and Structured Types</td>
<td>9</td>
</tr>
<tr>
<td>2.4 Pattern Matching</td>
<td>10</td>
</tr>
<tr>
<td>2.4.1 Overlapping Patterns</td>
<td>11</td>
</tr>
<tr>
<td>2.4.2 Nested Patterns</td>
<td>11</td>
</tr>
<tr>
<td>2.4.3 Multiple Arguments and Patterns</td>
<td>12</td>
</tr>
<tr>
<td>2.4.4 Semantics</td>
<td>14</td>
</tr>
<tr>
<td>2.4.5 Sum-Constructor Patterns</td>
<td>15</td>
</tr>
<tr>
<td>2.4.6 Product-Constructor Patterns</td>
<td>16</td>
</tr>
<tr>
<td>3 Graph Reduction</td>
<td>19</td>
</tr>
<tr>
<td>3.1 Overview</td>
<td>19</td>
</tr>
<tr>
<td>3.2 Representation and Redex Selection</td>
<td>19</td>
</tr>
<tr>
<td>3.2.1 Representation</td>
<td>19</td>
</tr>
<tr>
<td>3.2.2 REDEX Selection</td>
<td>20</td>
</tr>
<tr>
<td>3.3 Lambda Expressions</td>
<td>22</td>
</tr>
<tr>
<td>4 Modified Director String Calculus</td>
<td>28</td>
</tr>
<tr>
<td>4.1 Director String Calculus</td>
<td>28</td>
</tr>
<tr>
<td>4.1.1 Translation of Terms</td>
<td>31</td>
</tr>
<tr>
<td>4.2 Modified Director String Calculus</td>
<td>34</td>
</tr>
<tr>
<td>4.2.1 Translating (\lambda)-calculus to MDSC</td>
<td>36</td>
</tr>
<tr>
<td>4.3 An Algebra for MDSC</td>
<td>42</td>
</tr>
<tr>
<td>4.4 Conclusion and Future Work</td>
<td>55</td>
</tr>
<tr>
<td>Appendix A</td>
<td>56</td>
</tr>
<tr>
<td>A Church’s Numeral</td>
<td>56</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>60</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>A function application and λ abstraction.</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>Data Sharing</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>Standard Implementation of Y</td>
<td>24</td>
</tr>
<tr>
<td>3.4</td>
<td>Y Implemented Using Data Sharing</td>
<td>24</td>
</tr>
<tr>
<td>4.1</td>
<td>(λx. EF)G</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>λf. λx.f(f(x))</td>
<td>30</td>
</tr>
<tr>
<td>4.3</td>
<td>An arbitrary application node.</td>
<td>35</td>
</tr>
<tr>
<td>4.4</td>
<td>Two forms for λf.λx.f(fx)</td>
<td>38</td>
</tr>
<tr>
<td>4.5</td>
<td>λf.(λx.λy.x(yf))λp.p</td>
<td>39</td>
</tr>
<tr>
<td>4.6</td>
<td>A general application node</td>
<td>44</td>
</tr>
<tr>
<td>4.7</td>
<td>Rule I.1, LD = !, RD = !</td>
<td>45</td>
</tr>
<tr>
<td>4.8</td>
<td>Rule I.2, LD = !, RD = #</td>
<td>45</td>
</tr>
<tr>
<td>4.9</td>
<td>Rule I.3, LD = #, RD = !</td>
<td>45</td>
</tr>
<tr>
<td>4.10</td>
<td>Rule I.4, LD = #, RD = #</td>
<td>46</td>
</tr>
<tr>
<td>4.11</td>
<td>The result of reducing the redex at L0.</td>
<td>47</td>
</tr>
<tr>
<td>4.12</td>
<td>A unary node</td>
<td>48</td>
</tr>
<tr>
<td>4.13</td>
<td>Rule II.1, LD = !</td>
<td>48</td>
</tr>
<tr>
<td>4.14</td>
<td>Rule II.2, LD = #</td>
<td>49</td>
</tr>
<tr>
<td>4.15</td>
<td>A general collapsed application node.</td>
<td>49</td>
</tr>
<tr>
<td>4.16</td>
<td>λx.(λy.(xy))x</td>
<td>50</td>
</tr>
<tr>
<td>4.17</td>
<td>Reduction of λf.(λx.λy.x(yf))λp.p</td>
<td>53</td>
</tr>
</tbody>
</table>
Acknowledgments

First of all, I would like to thank my advisor Kazem Taghva for his patience, dedication and guidance throughout the course of my thesis work. He has provided me with insights into both the area of functional programming and theoretical computer science that I could not have gotten elsewhere. His constant support motivated me to explore topics that were new and intellectually challenging. I would like to thank Michael O’Donnell for his advise and Richard Kennaway for suggesting the new approach to director string calculus.

Thanks to Dr. Laxmi Gewali, Dr. John Minor and Dr. Ebrahim Salehi for serving on my committee.

A special thanks is also owed to Julie Borsack and Allen Condit for their friendship and encouragement; they made my graduate study more enjoyable. I would also like to acknowledge both Rick Lorber and Bill Jackson for their moral support.

Finally, I would like to thank my parents for giving me the inspiration to work to my full potential.
Chapter 1

Introduction

Overview

The fundamental computational model for functional programming languages is Lambda Calculus ($\lambda$-calculus) \cite{2, 3, 5, 6, 11, 9, 10, 12, 15, 16, 17, 20, 21} or a derivative of it such as combinatory logic. Many reduction systems have been proposed to evaluate $\lambda$-terms; these include Director String Calculus (DSC) proposed by Kennaway and Sleep \cite{13}. This system captures the essence of 'long reach' combinators by Turner \cite{26} while preserving the applicative structure of the original expression. The problem with the DSC was that it only captured weak reduction. To solve this problem, Sreedhar and Taghva \cite{29} introduced a 'Modified Director String Calculus' (MDSC) that captures strong reduction and preserves the applicative structure of the original expression. This system is complicated, in part, because of the case by case analysis used to develop a large set of rules. In this thesis we shall introduce a small set of rules, and an algebra suggested by Kennaway \cite{14} to manipulate directors and $\lambda$-expressions which eliminate the need for special cases.

We will explore the need for functional programming languages in the next section, and then give a short description of the important characteristics of modern functional languages. Chapter 2 will cover the compilation of functional programming languages into an enriched version of $\lambda$-calculus. In chapter 3 we will discuss proposed reduction
strategies for $\lambda$-expressions. In chapter 4, we will review the Director String Calculus (DSC), and introduce our algebra to demonstrate that it captures strong reduction.

**History of Computing**

In the early 1930's it was thought by Hilbert and others that every well defined mathematical problem could be solved algorithmically. Kurt Gödel published his incompleteness theorem in 1931 which informally states:

'There is no algorithm which we can use to decide if every mathematical sentence is true or false.'

Simply stated, we can always find a true number-theoretic statement that can be neither proved or disproved in a system. As a result of this theorem, people began trying to find systems that would characterize the set of all computable functions.

Alonzo Church invented $\lambda$-calculus in 1941 and stated the famous Church’s Thesis:

[4]

'Effectively computable functions from positive integers to positive integers are just those definable in the Lambda Calculus.'

Another characterization of computable objects was the 'Universal Machine' developed by Alan Turing. The Turing Machine is capable of modeling the same set of functions as $\lambda$-calculus [25]. The Turing Machine lead directly to the invention of the Von Neumann computer, which in turn was the precursor to several high level programming languages.

There are essentially two categories of programming languages, imperative and functional. Fortran, the first widely accepted imperative language, was created by Backus and others at IBM in the 1950's. The first functional language, LISP, was introduced by John McCarthy [18]; it appears to have been heavily influenced by
Another major area of study was the 'meaning' of programming languages. It is desirable to have a mathematical system to formalize the semantics of programming languages. Strachey and Scott used $\lambda$-calculus and its models to develop denotational semantics as a system for describing the meaning of programming languages [22, 24].

**Functional Languages**

The role of functional languages in computer science was profoundly influenced by Backus' 1977 ACM Turing Award Lecture [1]. Backus coined the phrase 'word-at-a-time' to describe the inherent problems associated with the Von Neumann bottleneck, and traditional imperative programming languages. Some of these problems are:

- 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 an entire set of statements to explicitly control program flow. The destructive assignment statement is a typical example of a command that causes side-effects.

- Because of side-effects, procedural programming is not *referentially transparent*. Generally, an expression cannot be replaced by its *value* without affecting the surrounding expressions.

- The focus when doing procedural programming is usually *how* to compute, rather than *what* to compute. The programmer has to know how the program gets evaluated at every step; this makes program and data abstraction difficult.

In an attempt to overcome some of these shortcomings, Backus introduced the functional language FP [1]. FP has a set of built-in functions, and it allows the programmer to combine functional forms to 'build' new functions and create programs.
Functions are used at a primitive level, which makes programs more mathematical. They can be understood more easily than those written in imperative languages. Furthermore, functional languages tend to be declarative, so what we are programming is emphasized instead of how to program. The mathematical nature of FP lead to an 'algebra of programs.' This algebra was used to derive laws that could be applied to show either program equivalence or program correctness.

A number of other functional programming languages have been developed including ISwim by Peter Landin [16] in the mid 1960's, ML and SML by Gordon et al. [19] at Edinburgh in 1979, as well as Turner's SASL, KRC, and Miranda\textsuperscript{1} [26, 27, 28]. While each of these languages have different features, each can be expressed in $\lambda$-calculus [8].

This evolution of functional languages has given us a set of distinguishing features which characterize them:

- **Higher Order Functions**: Functions are treated as first class values in a language; they may be stored in data structures, passed as arguments etc.

- **Lazy Evaluation**: Normal Order reduction rules of $\lambda$-calculus are used; they only evaluate an argument if it is needed and are guaranteed to find a normal form if one exists.

- **Abstraction Mechanism**: This allows the separations of a function definition from its use.

- **No side-effects**: There is no assignment statement in functional languages.

- **Referential Transparency**: An expression can be replaced by its value regardless of the surrounding expressions. This property makes it easier to prove program correctness, furthermore the order of evaluation does not affect the final result.

\textsuperscript{1}Miranda\textsuperscript{TM} is a trademark of Research Software Ltd.
• Equations and pattern-matching: Pattern matching and equations go together in functional languages. Since there are no side-effects, equational reasoning can be applied when evaluating functions. This will be explored further in chapter 2.

There are some drawbacks regarding functional languages:

• Compilers are quite inefficient when compared to the imperative counterparts on Von Neumann machines.

• The updating and handling of aggregate objects such as arrays is difficult.

The problem with aggregate data objects is that in functional languages declarations are generally for structures that are a single entity. With arrays, for example, a place holder must be used to keep place while updating the structure incrementally. This seems to contradict the style of functional programming that allows destructive updates.
Chapter 2
Compilation

Overview

In order to motivate the study of λ-calculus, this chapter will describe the compilation of functional languages into an enriched version of λ-calculus. Although the translation from enriched calculus to pure λ-calculus will not be discussed, the translations are straightforward [21, 8]. Program fragments will be presented in Miranda [28] where a program consists of a set of definitions and a set of expressions to be evaluated. The first section covers the translation of variables and functions. Subsequent sections will describe the compilation of structured types and pattern matching. The compilations presented are not necessarily the most efficient, but will suit our purposes to demonstrate the need for λ-calculus. ¹

Variables and Functions

Variables are defined in Miranda as using the ‘let’ command as in

\[ let \ x = 3 \ in \ x * 9 \]

Intuitively, this says to ‘give x the value of 3 in the expression x * 9.’ In λ-calculus, this becomes

\[ \lambda x. (\ x * 9 \) \ 3 \]

¹More efficient translations can be found in [21, Chapter 5, Philip Wadler].
Simple $\lambda$-calculus $\beta$-reduction is all that is required to evaluate this expression.

In general, the formula $let \ v = B \ in \ E$ is translated to $(\lambda v. \ E)B$. This translation also handles nested lets as well. For example, the expression,

$$let \ x = 10 \ in \ (let \ y = 12 \ in \ (+ \ x \ y))$$

$$\rightarrow \lambda x. \ ((\lambda y. \ (+ \ x \ y)) \ 12) \ 10$$

$$\rightarrow \lambda y. \ (+ \ (10 \ y)) \ 12$$

$$\rightarrow + \ (10 \ 12)$$

$$= 22$$

As a more useful example, look at the program fragment below:

```plaintext
square n = n * n
2 * (square 5)
```

First, the function

$$square \ n = n * n \rightarrow let \ square = \lambda n. \ n * n$$

Next, notice the expression $2 * (square 5)$ is what is being evaluated. Therefore, we would ultimately get the following:

$$let \ square = \lambda n. \ n * n$$

in $(2 * (square 5))$

A general function definition would be translated as follows:

$$f \ v_1 \ v_2 \ldots \ v_n = E$$

$$\equiv \ f = \lambda v_1. \ldots \lambda v_n. \ E$$

where the $v_i$'s are the arguments to the function. The 'letrec' (let recursively) command is used in the translation of function definitions similar to the way the let was used for variables. The general form of each is given below:
The general form for the letrec expression is:

\[
\text{letrec } v_1 = E_1, \quad v_2 = E_2, \quad \cdots \quad v_n = E_n
\]

\[
in \\ E
\]

The difference between the let and letrec expressions is that the \(v_i\) are in the scope of \(E_i\) and \(E\) in a letrec, but the \(v_i\) are only in the scope of the \(E\) in the let expression; the \(E_i\) are not influenced by \(v_i\)’s.

Consider the definition of factorial:

\[
\text{letrec factorial} = \lambda n. \text{IF } (= n 0) 1 * (n (\text{factorial}(-1 1)))
\]

\[
in \text{factorial 4}
\]

This defines a recursive function, factorial, applied to the argument 4. Notice that the factorial function is allowed to refer to itself by name—this is not normally allowed.
in pure λ-calculus. The implementation of recursion in λ-calculus requires the use of
the Y combinator which leads to the following solution:

\[
\text{letrec } v = B \text{ in } E \equiv (\text{ let } v = Y (\lambda v. B) \text{ in } E )
\]

The \text{letrec} is used in function definitions, since it is assumed that recursion is
always possible.

The following example will put the entire picture together:

\[
\begin{align*}
\text{average } a \ b & = (a + b) / 2 \\
\text{average } 2 \ (3+5) & \\
\text{First, consider the function definition:} & \\
\text{average } a \ b & = (a + b) / 2 \equiv \text{average } = \lambda a. \lambda b. (\ / \ ( + \ a \ b ) 2 ) \\
\text{Next, apply the letrec command:} & \\
\text{letrec} & \\
\text{average } = \lambda a. \lambda b. (\ / \ ( + \ a \ b ) 2) \text{ in} & \\
\text{average } 2 \ ( + \ 3 \ 5)) & \\
\end{align*}
\]

Finally, we must transform the expression into ordinary λ-calculus. For simplicity,
we will replace the \text{letrec} by a \text{let} since this definition is not recursive. This yields:

\[
( \lambda \text{average}. ( \text{average } 2 \ ( + \ 3 \ 5) ) ) \lambda a. \lambda b. (\ / \ ( + \ a \ b ) 2 )
\]

Constructors and Structured Types

Like their imperative counterparts, functional programming languages allow user de-
defined types. The mechanism used to declare them is the \textit{constructor}. Constructors
can be thought of as functions that ‘build’ user types and possibly assign types to the
variables created by the user. The language Miranda has several built in types like
integers, booleans, characters, lists and tuples. The following notation will be used
to describe them:
(x:xs) → CONS(x, xs)
(x:y:z:[]) → CONS(x, CONS(y, CONS(z, Nil)))
(3, True) → PAIR(num, bool)

Notice that pairs (a type of tuple) can have arguments of different types.

The user is free to build types from composites of built in types or composites of user defined types. To create a tree structure for storing integers, the following declarations would be used:

tree ::= LEAF num | BRANCH tree tree

LEAF and BRANCH are the constructors of the tree.

A function to exchange the values of the leaves of a tree would be written as:

reflect (LEAF n) = LEAF n
reflect (BRANCH t1 t2) = BRANCH (reflect t2) (reflect t1)

Function reflect behaves in the obvious way, if the argument passed to it is a leaf, then the value of the leaf is printed, otherwise, the function is called again with each subtree of the original branch with the left and right children exchanged. A function definition with patterns on the left side 'uses pattern-matching of its argument to do case analysis' [21, Page 52]. The main idea is that the arguments are evaluated only until their form is determined, then the correct operation of the function being applied can be executed. The remainder of this chapter will discuss the issues involved in pattern matching.

Pattern Matching

While the notion of pattern matching seems intuitively clear, a number of subtle issues arise. These include the need to handle overlapping patterns, nested patterns and patterns with multiple arguments. We will present several examples to illustrate these problems.
Overlapping Patterns

Consider the definition of the factorial function using Miranda. One obvious definition is:

\[
\begin{align*}
\text{fac} & \ 0 = 1 \\
\text{fac} & \ n = n \times \text{fac}(n - 1)
\end{align*}
\]

The equations are evaluated one at a time from top to bottom. In this case, the order of the equations is important, because the second equation would be executed for any integer value, so if the order of the equations was reversed, the one containing \( \text{fac} \ 0 \) would never be executed, and a nonterminating program would result. Functions whose arguments satisfy more than one equation are called overlapping.

One partial solution to this problem is to allow the use of guards to control the execution of functions. A guard is a conditional statement that evaluates the argument before allowing execution. Again consider the factorial function:

\[
\begin{align*}
\text{fac} \ n & = 1, \quad n = 0 \\
& = n \times \text{fac}(n - 1)
\end{align*}
\]

In this case, a single lefthand side controls the execution. The argument is evaluated, compared against the guards and the expression computed only when a guard is matched or a default case is found.

Nested Patterns

Another issue that must be addressed is that of allowing nested patterns. Consider a function to find the last element of a list:

\[
\begin{align*}
\text{lastElt}(x:[]) & = x \\
\text{lastElt}(x:xs) & = \text{lastElt}(xs)
\end{align*}
\]
The argument is evaluated to see if its 'tail' is equal to nil—if it is, then the last element has been found, otherwise the function lastElt must be called again. Also, notice that a call with an argument of nil would cause an error.

**Multiple Arguments and Patterns**

In general, functions have multiple arguments and multiple patterns. The following general form would characterize any function:

\[
\begin{align*}
    f \ p_1 & = E_1 \\
    f \ p_2 & = E_2 \\
    & \vdots \\
    f \ p_n & = E_n
\end{align*}
\]

where \(p_i\)'s are patterns.

We want the evaluation to proceed as 'try to match the first pattern \(p_1\) to the argument list, if that fails, try the second pattern and so on.' If a pattern match is found with pattern \(p_i\), then evaluate the expression \(E_i\); if no match occurs, an error is reported. This gives us the idea that a pattern match might fail; to handle this we will introduce a new constant, FAIL, to the enriched \(\lambda\)-calculus. Anytime an argument does not match a pattern, FAIL will be returned. We will also introduce a new infix operator \([\cdot]\). The \([\cdot]\) notation (pronounced 'FATBAR') is used to characterize the failure of a pattern match; it has the following semantics:

\[
\begin{align*}
    a [\cdot] b & = a, \quad \text{if } a \neq \bot \text{ and } a \neq \text{FAIL} \\
    \text{FAIL} [\cdot] b & = b \\
    \bot [\cdot] b & = \bot
\end{align*}
\]

where \(\bot\) represents 'bottom', a nonterminating computation.

To translate functions with multiple arguments, recall the translation for general functions, \(f \ v_1 \ v_2 \ldots v_n = E \equiv f = \lambda v_1 \ldots \lambda v_n. E\) where the \(v_i\)'s are arguments.
A similar translation for functions with multiple patterns would be: 

\[ f \ p_1 \ p_2 \ldots \ p_n = E \equiv f = \lambda p_1 \ldots \lambda p_n. \ E \]

where the \( p_i \)'s are patterns. To add multiple expressions, we would arrive at:

\[
\begin{align*}
  f \ p_1 &= E_1 \\
  f \ p_2 &= E_2 \\
  &\vdots \\
  f \ p_n &= E_n
\end{align*}
\]

which is then translated to:

\[
\begin{align*}
  f &= \lambda x. ( ((\lambda p_1. E_1)x) \\
      &\quad ((\lambda p_2. E_2)x) \\
      &\quad ((\lambda p_3. E_3)x) \\
      &\quad \vdots \\
      &\quad ((\lambda p_n. E_n)x) \\
      &\quad ERROR)
\end{align*}
\]

where \( x \) is the pattern of the arguments being evaluated.

Using these techniques on function \( reflect \) presented earlier would result in the following translation:

\[
\begin{align*}
  reflect &= \lambda t. ( ((\lambda (LEAF \ n). LEAF \ n) \ t) \\
            &\quad ((\lambda (BRANCH \ t1 \ t2). BRANCH( \ reflect(t2) \ reflect(t1))) \ t) \\
            &\quad ERROR)
\end{align*}
\]

where \( t \) is the pattern of the argument being evaluated. In this case, ERROR would never be returned because the only possible arguments to function \( reflect \) must be
either LEAF or BRANCH; any incorrect arguments would be caught by the type checker. However, it is not always the case that ERROR could not be returned. To illustrate this, look at the declaration of the function ‘head’ in Miranda.

\[
\text{head}(x : xs) = x \\
\equiv head = \lambda xs'. ((\lambda(CONS \ x \ xs). x)xs') \\
[] \ \text{ERROR}
\]

where \(xs'\) is the argument being evaluated. In this case, ERROR could occur since \(\text{head}(\text{Nil})\) is a legitimate error.

**Semantics**

We have defined the syntax of a set of pattern matching \(\lambda\)-abstractions; now we will define their semantics. We will discuss only Sum-Constructor Patterns and Product-Constructor Patterns in this section ([21, Pages 69-75] covers the topic in greater detail). A product type has only one constructor, while a sum type has two or more constructors. Pairs and triples are examples of product-constructors while color and boolean are examples of sum-constructors.

For purposes of notation, we will assume the existence of function \(EVAL\) which maps expressions to values (e.g. \(EVAL[5 + 2] = 7\)). To see the use of the function \(EVAL\), consider the semantics of ‘constant patterns’

\[
EVAL[\lambda k. \ E] \ a = \begin{cases} 
EVAL[\ E] & \text{if } a = EVAL[k] \\
FAIL & \text{if } a \neq \bot, \text{ and } a \neq EVAL[k] \\
\bot & \text{if } a = \bot
\end{cases}
\]

The first line compares argument \(a\) with the constant pattern \(k\), if they are the ‘same’, then the expression \(E\) is evaluated. The second line returns \(FAIL\), since argument \(a\)
is not the same as constant \( k \). The third line handles the case of a non-terminating program. The two examples below demonstrate these rules.

\[
(\lambda 1. + 3 4) 1 \rightarrow + 3 4
\]

\[
(\lambda 1. + 3 4) 2 \rightarrow \text{FAIL (since } 2 \neq 1)\]

**Sum-Constructor Patterns**

We will now discuss Sum-Constructor Patterns. We want to compare the arguments given to the function with the *constructor* used to build the object. When considering function \( \text{reflect} \),

\[
\text{reflect} (\text{LEAF } n) = \text{LEAF } n
\]

\[
\text{reflect} (\text{BRANCH } t_1 t_2) = \text{BRANCH} (\text{reflect } t_2) (\text{reflect } t_1)
\]

If the first argument to the function is of type LEAF, then we execute LEAF \( n \), and if it is of type BRANCH, then we execute the second line.

The semantics to evaluate sum-constructor patterns are as follows:

\[
EVAL[\lambda (s p_1 p_2 \ldots p_r). E](s a_1 a_2 \ldots a_r) =
\]

\[
EVAL[\lambda (p_1 p_2 \ldots p_r). E](a_1 a_2 \ldots a_r)
\]

\[
EVAL[\lambda (s p_1 p_2 \ldots p_r). E](s' a_1 a_2 \ldots a_r) = \text{FAIL } s \neq s'
\]

\[
EVAL[\lambda (p_1 p_2 \ldots p_r). E]\bot = \bot
\]

To apply these semantics to an argument \( A \), we evaluate \( A \) only to the point of determining what *constructor* it was formed with. If this form matches constructor form \( s \), then evaluate the remaining arguments \( (a_1 a_2 \ldots a_r) \) against patterns \( (p_1 p_2 \ldots p_r) \). Line two is ‘failure’, similar to the constant pattern case, and line three states that if the evaluation of the constructor form does not terminate, then neither does the function application in question. Again, let's look at function \( \text{reflect} \); recall
the function translation into enriched $\lambda$-calculus was:

\[
reflect = \lambda t. ( ((\lambda (\text{LEAF } n). \text{LEAF } n)t) \\
\quad [] ((\lambda (\text{BRANCH } t1 t2). \text{BRANCH}(reflect(t2)\text{reflect}(t1))t) \\
\quad [] \text{ERROR})
\]

where $t$ is the argument.

Consider an application of function $reflect$ with argument

$\text{BRANCH}(\text{LEAF } 1)(\text{LEAF } 2)$:

\[
\lambda((\text{LEAF } n). \text{LEAF } n) \text{BRANCH}(\text{LEAF } 1)(\text{LEAF } 2) = \text{FAIL}
\]

since $\text{LEAF } \neq \text{BRANCH}$ by rule 2.

Next, the $\text{BRANCH}$ constructor of $reflect$ must be evaluated,

\[
\lambda t1 \lambda t2. \text{BRANCH}(reflect(t2)\text{reflect}(t2)) (\text{LEAF } 0)(\text{LEAF } 1) \text{ by Rule 1.}
\]

\[
\rightarrow \lambda t2. \text{BRANCH}(reflect(t2)\text{reflect}(\text{LEAF } 0))(\text{LEAF } 1)
\]

\[
\rightarrow \text{BRANCH}(\text{LEAF } 1)(\text{LEAF } 0)
\]

**Product-Constructor Patterns**

Although a product type has only one constructor, the semantics of pattern matching are tricky. Recall that one of the characteristics of modern functional languages is *lazy evaluation*. This means that arguments are evaluated only if they are needed, so ideally we want a mechanism that binds the desired arguments to their corresponding formal parameter, but postpones evaluating them until they are needed. Consider the following family of ‘zero’ functions:

\[
\text{ZeroAny } x = 0
\]

\[
\text{ZeroList}[ ] = 0
\]
ZeroPair(x,y) = 0

The function ZeroAny is clearly lazy, while the function ZeroList is strict (i.e. always evaluates its argument), since its argument must be equal to ‘Nil.’ At first glance, it appears that ZeroPair might need to evaluate its argument, but this is not the case. The type checker would issue an error if the argument to ZeroPair was not of the PAIR type, and after that the function does not need the argument. The significance of lazy product matching is that using lazy semantics, ZeroPair ⊥=0, while under strict semantics, ZeroPair ⊥=⊥.

The semantics of product-constructor patterns is given below:

\[
\text{EVAL}\[ \lambda( t \ p_1 \ p_2 \ldots \ p_r ). E ] a = \\
\text{EVAL}\[ \lambda p_1 \ p_2 \ldots \ p_r ). E \] (SEL-t-1 a) (SEL-t-2 a) \ldots (SEL-t-r a)
\]

where SEL-t-i is a new function,

\[
\text{SEL-t-i( t \ a_1 \ a_2 \ldots a_r )} = a_i \\
\text{SEL-t-i \ ⊥} = ⊥
\]

The evaluation of arguments is postponed by binding the names for the components needed to applications of SEL-t-i rather than evaluating the argument immediately. This implies that all elements of an argument list are bound to formal parameters, but they are only evaluated when they are needed. Consider the following example using function ZeroPair:

\[
\text{ZeroPair} = \lambda(\text{PAIR} \ X \ Y). 0 \quad \text{by function translation}
\]

\[
\text{EVAL}[ \text{ZeroPair} ] \bot \\
= \text{EVAL}[ \lambda(\text{PAIR} \ x \ y ). 0 ] \bot
\]
\[ \text{EVAL}[ \lambda x \lambda y . 0 ] (\text{SEL-PAIR-1 } \bot ) (\text{SEL-PAIR-2 } \bot ) \]
\[ \rightarrow \text{EVAL}[ \lambda y . 0 ] (\text{SEL-PAIR-2 } \bot ) \]
\[ \rightarrow 0 \]

The last two evaluations are done using \( \beta \)-reduction; the argument itself (\( \bot \)) was never evaluated, it was merely bound to function SEL-PAIR-i in case it was needed.

Lazy semantics have the advantage that they eliminate unnecessary argument evaluations. They also allow the computation of things that are infinite like lists of integers (1, 2, 3 ...). Unfortunately, these benefits carry an implementation cost which will be discussed below:

Consider the function \( \text{AddPair} = \lambda (\text{PAIR } x \ y). + x \ y \) which adds the elements of a pair together. Let's look at the evaluation of a function call \( \text{AddPair}(\text{PAIR } 3 \ 4) \).

\[
\text{AddPair}(\text{PAIR } 3 \ 4) = ( \lambda (\text{PAIR } x \ y) (\text{PAIR } 3 \ 4)) \\
= ( \lambda x \ y. + x \ y) \text{SEL-PAIR-1}(\text{PAIR } 3 \ 4) \text{SEL-PAIR-2}(\text{PAIR } 3 \ 4) \\
\rightarrow ( \lambda y + (\text{SEL-PAIR-1}(\text{PAIR } 3 \ 4) \ y) \text{SEL-PAIR-2}(\text{PAIR } 3 \ 4) \\
\rightarrow + (\text{SEL-PAIR-1}(\text{PAIR } 3 \ 4) \text{SEL-PAIR-2}(\text{PAIR } 3 \ 4)) \\
\rightarrow + (3 (\text{SEL-PAIR-2}(\text{PAIR } 3 \ 4))) \\
\rightarrow + (3 \ 4) \\
= 7
\]

If strict semantics were used, the following would be the result:

\[
\text{AddPair}(\text{PAIR } 3 \ 4) = ( \lambda (\text{PAIR } x \ y) (\text{PAIR } 3 \ 4)) \\
= ( \lambda x \ y. + x \ y) (3 \ 4) \\
\rightarrow ( \lambda y + 3 \ y) 4 \\
\rightarrow ( + 3 \ 4 ) \\
= 7
\]

When all arguments to a function are needed, there is a cost involved in implementing lazy semantics. The study of this area is called \textit{strictness analysis}. 
Chapter 3

Graph Reduction

Overview

This chapter will discuss the concept of Graph Reduction. It provides both a representation for $\lambda$-expressions, and a mechanism for 'processing' them. The reduction mechanism must be faithful to the properties of $\beta$-reduction within $\lambda$-calculus; to guarantee that the result will meet the properties of Church-Rosser. We will discuss the conceptual representation of $\lambda$-expressions, the methods and choices involved in selecting the next reducible expression (redex), and the transformations of $\lambda$-expressions necessary to implement 'Supercombinators.' [2, 11]

Representation and Redex Selection

Representation

If we assume the successful translation of the given functional program into a set of $\lambda$-expressions, then we must have a representation for these $\lambda$-terms. The generally accepted representation is abstract syntax trees where the leaves of the trees are functions or constants, and the $@$ symbol represents function application. An example is shown in figure 3.1. The selection of trees as a representation also allows the internal data structures for storing the expressions to be straightforward.
REDEX Selection

When choosing a reduction strategy, one of the most important issues is determining the order of subexpression evaluation. Typically, expressions in programs are made of subexpressions, and the order of the subexpression evaluation can have significant impacts on performance. For instance, it is critical to decide if the evaluation of arguments passed to functions will take place upon the call to the function, or be postponed until the arguments are needed. As mentioned in the previous chapter, most modern functional languages have adopted the lazy approach which postpones this evaluation. This is similar to the ‘call by need’ technique employed in Algol, but unfortunately, this concept is difficult to implement in imperative languages.

Consider the call to a function in an imperative language, because the evaluation of arguments may produce side effects which the function may depend upon, arguments must be evaluated even though they may not be used by the function being called.

The order of reduction proposed most often in the literature is called Normal Order Reduction. ¹ This method is convenient when implementing lazy semantics, because arguments to functions are only evaluated when needed. The use of data sharing will guarantee that they are evaluated at most once, since additional uses of the argument will use the value computed the first time. Normal Order Reduction states that the leftmost outermost redex should be reduced first. To clarify this meaning, consider a

¹Normal order reduction is optimal for SK Machines [23], but it is not known whether this is optimal for graph reduction techniques [17]
function application. The outermost redex is the function itself, so the normal order reduction will reduce the application of the function to its argument prior to reducing the argument. For instance,

\((\lambda x. 3) \perp \rightarrow 3\) where \(\perp\) is a nonterminating computation

This is a function that returns '3' regardless of its argument, so the argument is never evaluated. In general, we should stop reducing an expression in a graph even though there may be redexes left in the graph. More formally stated,

'Do normal reduction, but stop when there is no top-level redex even though there may be inner redexes in the graph.'

To state this mathematically, we need the following definition:

A \(\lambda\)-expression is in **Weak Head Normal Form** (WHNF) if and only if it is of the form

\[ F \, E_1 \, E_2 \ldots \, E_n \text{ where } n \geq 0 \]

and

- \(F\) is a variable or data object or
- \(F\) is a \(\lambda\)-abstraction, or built-in function and
- \(F \, E_1 \, E_2 \ldots \, E_m\) is not a redex for any \(m \leq n\).

An expression has no top level redex iff it is in WHNF.

Let us first consider the evaluation of built-in functions. If the evaluator finds that the top level redex is an application of a built-in function which evaluates its arguments, it has to check whether the arguments are also in WHNF. If they are not, then they must be reduced before the function can be called. For instance, in the expression

\[ \text{IF (NOT TRUE) f g h} \]
the redex is \( \text{IF (NOT TRUE)} \ f \ g \). The \text{IF} function must evaluate its argument (NOT TRUE) which leads to FALSE; then the evaluation of the redex would continue as:

\[
\text{IF (FALSE)} \ f \ g = g, \text{ so the original expression}
\]

\[
\text{IF (NOT TRUE)} \ f \ g \ h
\]

would evaluate to \( g \ h \)

The obvious question that comes to mind is 'How do we find the next top-level redex?' If we have an expression of the form \( F \ E_1 \ E_2 \ldots \ E_n \)

where the \( E_i \)'s represent zero or more arbitrarily complicated arguments. We know that \( F \) is one of the following:

1. \( F \) is a data object (like CONS). In this case the expression is already in WHNF.
   Also, if \( n \neq 0 \) we are trying to apply a data object to an argument which is an error.

2. \( F \) is a built-in function of \( k \) arguments. If there are enough arguments, \( (n \geq k) \),
   then \( F \ E_1 \ E_2 \ldots \ E_k \) is the outermost redex which normal order chooses.
   If \( n < k \), then the expression is already in WHNF.

3. \( F \) is a \( \lambda \)-abstraction.
   If \( n > 0 \) then \( F \ E_1 \) is the next argument to reduce.

**Lambda Expressions**

The most common element we must compile is the \( \lambda \)-expression. In doing so, we begin with the abstract syntax tree for the original expression, and perform transformations on it until an acceptable normal form is reached. In the interest of lazy evaluation rules, it should be clear that expressions are only evaluated once; subsequent uses of the same expression should use the first value computed. This leads to three important ideas:
1. There is a need for shared data [30].

2. We should overwrite the root of the redex with the final result of the expression being evaluated.

3. λ-abstractions themselves may be shared, so we must construct a new instance of the λ-body rather than substituting the original λ-body directly.

The combinations of sharing data, and overwriting redexes accomplishes the goal of evaluating expressions only once (see figure 3.2). The notions of data sharing and overwriting the root of the redex also lead to a more efficient implementation of recursion. Recall that recursion is implemented using the Y Combinator as

\[ Yf \rightarrow f(Yf) \]

which is shown in figure 3.3. By overwriting the root of the redex, recursion can be implemented as:

\[ Yf \rightarrow f(Yf) \]
\[ \rightarrow f(f(Yf)) \]
which is shown in figure 3.4.

The final item to be addressed is the construction of new instances of a λ-body. When doing this construction, it is desirable to allow easy substitution for the formal parameters within the abstraction. We will transform the λ-expression into supercombinators using a process known as lambda-lifting [11]. Unfortunately, the transformation to supercombinators does not work for all forms of λ-expressions as we will see from the following definition:

A supercombinator, S, of arity n is a λ-expression of the form

\[ \lambda x_1.\lambda x_2 \ldots \lambda x_n. E \]
where \( E \) is not a \( \lambda \)-abstraction such that

1. \( S \) has no free variables
2. Any \( \lambda \)-abstraction in \( E \) is also a supercombinator
3. \( n \geq 0 \)

The fact that supercombinators have no free variables allows us to compile them into a fixed code sequence. Also, item 2 above guarantees that no \( \lambda \)-abstraction in the body contains free variables, so there is no need to copy them when instantiating the supercombinator body.

The problem is that supercombinators cannot contain free variables, while general \( \lambda \)-expressions may contain them. To transform \( \lambda \)-expressions containing free variables to supercombinators, \( \beta \)-abstraction is used. Consider the following example:

\[
(\lambda x. (\lambda y. + y x) x) 4
\]

In this case neither of the \( \lambda \)-abstractions is a supercombinator.

The \( x \) is free in \((\lambda y. + y x)\), so \( \beta \)-abstraction is required.

This yields, \((\lambda w. \lambda y. + y w) x\)

where \( w \) is the new variable.

Substituting this into the original expression gives us:

\[
(\lambda x. (\lambda w. \lambda y. + y w) x x) 4
\]

Now the above expression is a supercombinator and we can proceed as follows:

A supercombinator redex consists of the application of a supercombinator to \( n \) arguments. The reduction replaces the redex by an instance of the supercombinator body with the arguments substituted for the corresponding formal parameter.
The application of the supercombinator to n arguments is accomplished by naming the supercombinator, and then calling it with the appropriate parameters. Let the inner supercombinator above be named $\mathcal{Y}$, then the code sequence above becomes:

$$\mathcal{Y} w y = + y w$$

$$\left( \lambda x. \mathcal{Y} x x \right) 4$$

Notice that the remaining $\lambda$-expression is also a supercombinator, so it can also be named (say $\mathcal{X}$) and then 'called.' This give us

$$\mathcal{X} x = \mathcal{Y} x x$$

Finally, the entire expression becomes

$$\mathcal{Y} w y = + y w$$

$$\mathcal{X} x = \mathcal{Y} x x$$

$$\mathcal{X} 4$$

The full evaluation would be:

$$\mathcal{X} 4$$

$$= \mathcal{Y} 4 4$$

$$= + (4 4)$$

$$= 8$$

For each $\lambda$-abstraction, we perform the following steps:

1. Choose any $\lambda$-abstraction with no inner $\lambda$-abstractions in its body.

2. Remove free variables from $\lambda$-abstractions as parameters to allow the transformation into supercombinators.
3. Name the resulting supercombinators.

4. Replace the occurrences of the $\lambda$-abstraction by the name of the supercombinator applied to the free variables.

5. Compile the $\lambda$-abstraction and associated name into 'fixed code.'
Chapter 4

Modified Director String Calculus

Director String Calculus

Kennaway and Sleep [13] introduced Director String Calculus (DSC) as an alternate representation for $\lambda$-abstractions. The idea is to extend the abstract tree representation in which each application node has associated with it a 'string of directors' and each director is associated with a level of abstraction. The director strings encode the distribution of formal parameters to the subtrees in an abstraction. For instance, consider the evaluation of the expression $(\lambda x. EF)G$ depicted in figure 4.1. The substitution of $G$ for $x$ ($x := G$) will be conveyed to both $E$ and $F$, to $E$ alone, to $F$ alone or to neither $E$ nor $F$ depending upon the free occurrences of $x$ in both $E$ and $F$. To define this system more formally, we will need a set of terms, and sets of rules.

![Diagram of $(\lambda x. EF)G$]

Figure 4.1: $(\lambda x. EF)G$
to translate terms back and forth from \( \lambda \)-terms to director string terms.

Let \( \text{VAR} \) be an infinite set of variables, then the set of terms, denoted by \( \Lambda \), is defined as follows:

- \( v \) is a term, where \( v \in \text{VAR} \)
- \( (MN) \) is a term where \( M, N \in \Lambda \)
- \( \lambda v.M \) is a term, where \( M \in \Lambda \) and \( v \in \text{VAR} \)

Also, let the set of directors, \( \{ \wedge, /, \backslash, - \} \), have the following meanings for distribution of formal parameters:

- \( \wedge \Rightarrow \) Send to both the left and right
- \( / \Rightarrow \) Send only to the left, not to the right
- \( \backslash \Rightarrow \) Send only to the right, but not to the left
- \( - \Rightarrow \) Send neither to the right nor to the left

These symbols are used to define a set of rules which abstract a variable from an application:

1. \( \lambda x.(E_x F_x) \rightarrow \wedge(\lambda x. E_x)(\lambda x. F_x) \)
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. \( \lambda x.(EF) \rightarrow -(EF) \)
5. \( \lambda x. x \rightarrow I \)
6. \( \lambda y. x \rightarrow Ky \)

where \( E_x \) represents the freeness of \( x \) in \( E \). The last two rules are needed for simple variables where \( K \) and \( I \) are the standard combinators [5].
The use of these rules can be described as moving an abstraction through an application leaving a director symbol behind and transmitting the abstraction process to the right or left as appropriate. For instance, the second rule above can be read as:

'The abstraction of \( \text{x} \) from a combination in which \( \text{x} \) occurs free on the left, but not on the right can be encoded as send to the left followed by the application of the left side (which contains \( \text{x} \)), with the right side (which does not).'

Consider the expression \( \lambda f. \lambda x. f(f(x)) \) as shown in figure 4.2 beginning with the innermost \( \lambda \)-term \((\text{x})\).

- Notice that \( \text{x} \) goes right at the top level, so the first director is \( \backslash \).
- At the next level, the \( \text{x} \) is also on the right, again the director is \( \backslash \).
- Moving back to the top level, \( f \) is on both the left and the right, so use \( \wedge \).
- At the second level, \( f \) appears on the left, but not on the right, use \( / \).

The following will define the set of director string terms:
1. \(\{\wedge, /, \backslash, -\}\) are the binary directors. The set of words over this alphabet is denoted by \(DIR_2\).

2. \(\{!, \#\}\) are the unary directors. The set of words over this alphabet is denoted by \(DIR_1\).

3. \(\{\Delta\}\) is a hole or place holder.

4. \(x, y, z, \ldots\) are an infinite list of variables denoted by \(VAR\)

# and ! are called **discard** and **insert** respectively. Also, let

- \(D\) = The arbitrary director string
- \(D_1\) = The arbitrary unary director string
- \(D_2\) = The arbitrary binary director string
- \(d\) = The arbitrary director symbol
- \(d_1\) = The arbitrary unary director symbol
- \(d_2\) = The arbitrary binary director symbol
- \(@_1\) = The empty unary director string
- \(@_2\) = The empty binary director string

Let \(ATOMS = VAR \cup \{\Delta\}\), then the set of Director String Terms \((DST)\) is defined as:

- \(a \in ATOMS \Rightarrow (D_1, a) \in DST\)
- \(E \in DST \Rightarrow (D_1, E) \in DST\)
- \(E_1, E_2 \in DST \Rightarrow (D_2, E_1, E_2) \in DST\)

**Translation of Terms**

We need systems to translate terms back and forth from \(\lambda\)-calculus to director string calculus. To assist in this translation, we define the set of Director String Lambda
Terms (DS\(\lambda\)T) as follows:
\[
E \in DST \Rightarrow E \in DS\lambda T
\]
\[
x \in VAR, E \in DST \Rightarrow \lambda x.E \in DS\lambda T
\]
To map the set of \(\lambda\)-calculus terms to DS\(\lambda\) terms,

Replace every occurrence of a variable \(v\) by \((@_1, v)\)

Replace every application \((E F)\) by \((@_2, E, F)\)

The DS\(\lambda\) terms are then translated into \(DST\) using the following rules:

1. \(\lambda x.(D_1, x) \rightarrow (\!D_1, \Delta)\)
2. \(\lambda x.(D_1, E) \rightarrow (\#D_1, E)\)
3. \(\lambda x.(D_2, E\_x, F\_x) \rightarrow (\wedge D_2, (\lambda x.E\_x), (\lambda x.F\_x))\)
4. \(\lambda x.(D_2, E\_x, F) \rightarrow (\lor D_2, (\lambda x.E\_x), F)\)
5. \(\lambda x.(D_2, E, F\_x) \rightarrow (\Diamond D_2, E, (\lambda x.F\_x))\)
6. \(\lambda x.(D_2, E, F) \rightarrow (\neg D_2, E, F)\)

For example,
\[
\lambda f.\lambda x.(f(fx)) \rightarrow \lambda f.\lambda x.(@_2, (@_1, f), (@_2, (@_1, f), (@_1, x)))
\rightarrow \lambda f.(\backslash, (@_1, f), (\lambda x.(@_2, (@_1, f), (@_1, x))))
\rightarrow \lambda f.(\backslash, (@_1, f), (\lambda, (@_1, f), \lambda x.(@_1, x)))
\rightarrow \lambda f.(\backslash, (@_1, f), (\lambda, (@_1, f), ![\Delta]))
\rightarrow (\wedge \backslash, \lambda f.(@_1, \lambda f.(\backslash, (@_1, f), ![\Delta])))
\rightarrow (\wedge \backslash,[\Delta], (\backslash, \lambda f.(@_1, f), ![\Delta]))
\rightarrow (\wedge \backslash,[\Delta], (\backslash, ![\Delta, ![\Delta]]))
\]
where $!A$ replaces $I$.

A set of rules is also needed to convert DS Terms to $\lambda$-terms.

1. $(!D, (\lambda x. E)) \rightarrow \lambda x.(D, E)$

2. $(\#D, E) \rightarrow \lambda x.(D, E)$ ($x$ is a new variable not occurring in $E$)

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

4. $(/D, (\lambda x. E), F) \rightarrow \lambda x.(D, E, F)$

5. $(\land D, E, (\lambda x. F)) \rightarrow \lambda x.(D, E, F)$

6. $(-D, E, F) \rightarrow \lambda x.(D, E, F)$ ($x$ is a new variable)

7. $\Delta \rightarrow \lambda x.x$

The rules involving $\land$ and $/$ cause no variable clash since it is assumed that $\#$, $\land$, and $\Delta$ create new variables. To use these rules, begin with the $\Delta$ terms and create a new, distinct variable for each. This introduces $\lambda$'s which allow other rules to be used. When the rules involving $\lambda$'s are used, the $\lambda$'s make their way up the structure. A pair of $\lambda$'s appearing at the children of a binary director string beginning with a $\land$ are merged.

The final item to be addressed is the evaluation of DS Terms.

1. $(\otimes_2, (\land D, E_1, E_2), E_3) \rightarrow (D, (\otimes_2, E_1, E_3), (\otimes_2, E_2, E_3))$

2. $(\otimes_2, (/D, E_1, E_2), E_3) \rightarrow (D, (\otimes_2, E_1, E_3), E_2)$

3. $(\otimes_2, (-D, E_1, E_2), E_3) \rightarrow (D, E_1, (\otimes_2, E_2, E_3))$

4. $(\otimes_2, (\#D, E_1, E_2), E_3) \rightarrow (D, E_1, E_2)$

5. $(\otimes_2, (!D, E_1, E_2) \rightarrow (D, (\otimes_2, E_1, E_2))$
6. \((\varnothing_2, (\#D, E_1), E_2) \rightarrow (D, E_1)\)

7. \((\varnothing_2, \Delta, E_1) \rightarrow E_1\)

These conversion rules are activated by the empty directors \(\varnothing_2\) and \(\varnothing_1\); each is interpreted as ‘transmit argument.’ For instance, the first rule states transmit \(E_3\) to both \(E_1\) and \(E_2\).

Kennaway and Sleep state that the \(\lambda\)-calculus based on DSC is correct up to \(\beta\)-convertability. They also state that for terms whose normal form contains no \(\lambda\)'s (called ground terms), their system yields strong reduction. Strong reduction implies that the normal form obtained by DSC is the same as the normal form obtained when using pure \(\lambda\)-calculus. Also it is important to note that the applicative structure of the original expression is preserved; this guarantees that the number of subterms in the translation cannot be greater than the number of subterms in the original expression.

**Modified Director String Calculus**

To see that the director string calculus does not capture strong reduction, consider the following term:

\[
\lambda x.(\lambda y.xy)x \rightarrow (\lambda, (/, (\Delta), (\Delta), (\Delta)))
\]

The translated term on the right is in normal form in DSC, but its normal form in \(\lambda\)-calculus is \(\lambda x. xx\). One problem is the loss of information during the transformation of \(\lambda\)-terms to DS terms described in the previous section; figure 4.3 will clarify this idea. Suppose, for example that \(d=\;/\). This says to send the argument to the left, but it does not explicitly state not to send the argument to the right. Without this information, further reduction is not possible; this information is kept in the Modified Director String Calculus (MDSC) developed by Sreedhar and Taghva [29].
that captures strong reduction. We will enhance this system by adding an algebra which simplifies the case-by-case analysis, and still captures strong reduction.

Let \( \# = \text{discard} \), \( ! = \text{transmit} \) and : be the delimeter between the left and right parts of the director string, then director strings are split into their left and right components as follows:

\[\begin{align*}
\wedge & \text{ will be replaced by } ! : ! \\
/ & \text{ will be replaced by } ! : \# \\
\backslash & \text{ will be replaced by } \# : ! \\
- & \text{ will be replaced by } \# : \#
\end{align*}\]

For instance, \( \backslash \) which was 'send to the right' is replaced by \( \# : ! \) where the \( \# \) says 'do not send to the left' and the \( ! \) means 'send to the right' since it is on the right side of the delimeter symbol (:). When considering a function application, each incoming argument has a left and a right director symbol associated with it to define its movement down the tree. The string on the left of the delimeter specifies which incoming arguments are passed to the left, and the right hand string specifies which arguments are sent to the right.

The set of modified director string terms (MDS Terms) is defined as follows:

1. \( !, \# \) are directors. The set of words over the alphabet of binary directors is denoted by \( DIR_2 \). The set of words over the alphabet of unary directors is
denoted by $DIR_1$.

2. $\Delta$ is a place holder or hole.

3. $x, y, z...$ an infinite set of variables denoted by $VAR$.

As with DSC, let

$$D = \text{The arbitrary director string}$$

$$D_1 = \text{The arbitrary unary director string}$$

$$D_2 = \text{The arbitrary binary director string}$$

$$d = \text{The arbitrary director symbol}$$

$$d_1 = \text{The arbitrary unary director symbol}$$

$$d_2 = \text{The arbitrary binary director symbol}$$

$$\emptyset_1 = \text{The empty unary director string}$$

$$\emptyset_2 = \text{The empty binary director string}$$

$$ATOMS = VAR \cup \{\Delta\}$$

The set of MDS terms is defined to be:

$$a \in ATOMS \Rightarrow (D_1, a) \in MDST$$

$$E \in MDST \Rightarrow (D_1, E) \in MDST$$

$$E_1, E_2 \in MDST \Rightarrow (D_2, E_1, E_2) \in MDST$$

**Translating $\lambda$-calculus to MDSC**

We need a system to translate terms back and forth from $\lambda$-calculus to MDSC. To assist in this translation, we define the set of Modified Director String Lambda Terms (MDS$\lambda$T) as follows:

1. $E \in MDST \Rightarrow E \in MDS\lambda T$

2. $x \in VAR, E \in MDST \Rightarrow \lambda x.E \in MDS\lambda T$
If we begin with a set of $\lambda$-calculus terms, then they can be mapped into MDS$\lambda$ terms as follows:

1. Replace every occurrence of a variable $v$ by ($@_1, v$)
2. Replace every application $(E F)$ by ($@_2, E, F$)

We can now define a set of rules to go from terms in MDS$\lambda$T to MDST which removes all $\lambda$'s and bound variables. Let $D_2$ denote a director string split into its left and right parts as $L : R$, then the notation $d_a : d_b D_2$ represents $d_a L : d_b R$ where $d_a, d_b \in \{!, #\}$.

1. $\lambda x. (D_1, x) \rightarrow (\# D_1, \Delta)$
2. $\lambda x. (D_1, \Delta) \rightarrow (\# D_1, \Delta)$
3. $\lambda x. (D_1, y) \rightarrow (\# D_1, y)$ if $x \neq y$
4. $\lambda x. (D_2, E_2, F_2) \rightarrow (\# : ! D_2, (\lambda x. E_2), (\lambda x. F_2))$
5. $\lambda x. (D_2, E_2, F) \rightarrow (\# : ! D_2, (\lambda x. E_2), (\lambda x. F))$
6. $\lambda x. (D_2, E, F_2) \rightarrow (\# : ! D_2, (\lambda x. E), (\lambda x. F_2))$
7. $\lambda x. (D_2, E, F) \rightarrow (\# : \# D_2, (\lambda x. E), (\lambda x. F))$

Using this system,

\[ \lambda f. \lambda x. (f(fx)) \rightarrow \lambda f. \lambda x. (\@_2, (\@_1, f), (\@_2, (\@_1, f), (\@_1, x))) \]
\[ \rightarrow \lambda f. (\# : !, \lambda x. (\@_1, f), \lambda x. (\@_2, (\@_1, f), (\@_1, x))) \]
\[ \rightarrow \lambda f. (\# : !, (\#, f), (\# : !, \lambda x. (\@_1, f), \lambda x. (\@_1, x))) \]
\[ \rightarrow \lambda f. (\# : !, (\#, f), (\# : !, (\#, f), (!, \Delta))) \]
\[ \rightarrow (\# : !, \lambda f. (\#, f), \lambda f. (\# : !, (\#, f), (!, \Delta))) \]
\[ \rightarrow (\# : !, (\#, \Delta), (\# : \#, \Delta), \lambda f. (\#, f), !, \Delta)) \]
\[ \rightarrow (\# : !, (\#, \Delta), (\# : \#, \Delta), (\# : \#, \Delta), (!, \Delta)) \]
Observe that the director string associated with $\Delta$ is also occurring on the left or right of the : depending on whether $\Delta$ is the right or left operand. Therefore, we can represent the last expression as:

$$(!# : !!, \Delta, (!# : #!, \Delta, \Delta))$$

Pictorially this is either graph in figure 4.4. We should also point out that this observation will not hold when we start applying our reduction rules. Let $D$ below have the same definition as $D_2$ above, then to translate MDS terms back to $\lambda$-terms:

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. $(! : ! D, \lambda x.E, \lambda y.F) \rightarrow \lambda z.(D, E[x:=z], F[y:=z])$, where $z$ is a new variable
5. $(! : # D, \lambda x.E, \lambda y.F) \rightarrow \lambda z.(D, E[x:=z], F)$, where $z$ is a new variable
6. $(# : ! D, \lambda x.E, \lambda y.F) \rightarrow \lambda z.(D, E[F[y:=z]])$, where $z$ is a new variable
7. $(# : # D, \lambda x.E, \lambda y.F) \rightarrow \lambda z.(D, E, F)$ 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. $(\oplus_2, E, F) \rightarrow (EF)$

Consider the term $\lambda f.(\lambda x.\lambda y.x(yf))\lambda p.p$ as shown in figure 4.5. In this example, we will convert the MDSC expression back into $\lambda$-calculus. As seen in the tree representation, the MDSC term is:

\[(! : \#, (!\# : !\#, (!\# : \Delta), (!\# : \Delta), (!\# : \Delta)), (!\#, \Delta))\]

To translate this expression into $\lambda$-calculus, begin with the simple subterms:

\[(!\# : \Delta) \rightarrow (!\#, \lambda x_1.x_1) \text{ by 1}\]
\[\rightarrow \lambda x_2(!\#, \lambda x_1.x_1) \text{ by 3}\]
\[\rightarrow \lambda x_2.\lambda x_1.(\#, x_1) \text{ by 2}\]
Now continuing with the subterm $(!\#\#, \Delta)$,

\[
(!\#\#, \Delta) \rightarrow (\#\#, \lambda x_4. x_4) \text{ by 1}
\]
\[
\rightarrow \lambda x_4. (\#\#, x_4) \text{ by 2}
\]
\[
\rightarrow \lambda x_4. \lambda x_5. (\#, x_4) \text{ by 3}
\]
\[
\rightarrow \lambda x_4. \lambda x_5. \lambda x_6. (\@_1, x_4) \text{ by 3}
\]

Moving on to expression $(\#\#, \Delta)$,

\[
(\#\#, \Delta) \rightarrow (\#, \lambda x_7. x_7) \text{ by 1}
\]
\[
\rightarrow \lambda x_8. (\#, \lambda x_7. x_7) \text{ by 3}
\]
\[
\rightarrow \lambda x_8. \lambda x_9. (!, \lambda x_7. x_7) \text{ by 3}
\]
\[
\rightarrow \lambda x_8. \lambda x_9. \lambda x_7. (\@_1, x_7) \text{ by 2}
\]

The final subterm is $(!\#, \Delta)$

\[
(\#, \Delta) \rightarrow (\#, \lambda x_{10}. x_{10}) \text{ by 1}
\]
\[
\rightarrow \lambda x_{11}. (!, \lambda x_{10}. x_{10}) \text{ by 3}
\]
\[
\rightarrow \lambda x_{11}. \lambda x_{10}. (\@_1, x_{10}) \text{ by 2}
\]

Using substitution, consider the innermost subterm, $(\#\# : \#\#, (\#\#, \Delta), (!\#\#, \Delta))$, (let this be equal to $Q$),

\[
Q \rightarrow (\#\# : \#\#, \lambda x_8. \lambda x_9. \lambda x_7. (\@_1, x_7), \lambda x_4. \lambda x_5. \lambda x_6. (\@_1, x_4))
\]
Looking back at the original expression,

\((! : \#, (#! : \#!, (#! : \#!, \Delta), (#! : \#!, (## : \#!), (## : \#!, \Delta), (## : \#!, \Delta)) we have \(\lambda\)-expressions for each of the underlined subexpressions. Substitution yields:

\[
\begin{align*}
(\#! : \#!, (\lambda x_2. \lambda x_3. (\tau_1, x_1), \lambda x_{12}. \lambda x_{13}. \lambda x_{14}. (\tau_2, (\tau_1, x_{14}), (\tau_1, x_{12})))
\rightarrow \lambda x_{15}. ((\# : \#!, (\lambda x_1. \lambda x_3. (\tau_1, x_1), \lambda x_{13}. \lambda x_{14}. (\tau_2, (\tau_1, x_{14}), (\tau_1, x_{15}))) by 6
\rightarrow \lambda x_{15}. \lambda x_{16}. (\# : !, (\lambda x_3. (\tau_1, x_{16}), \lambda x_{14}. (\tau_2, (\tau_1, x_{14}), (\tau_1, x_{15}))) by 5
\rightarrow \lambda x_{15}. \lambda x_{16}. \lambda x_{17}. (\tau_2, (\tau_1, x_{17}), (\tau_1, x_{15}))) by 6
\end{align*}
\]

This gives us

\[
\begin{align*}
(\#!, (\lambda x_{16}. \lambda x_{17}. \lambda x_{18}. (\tau_2, (\tau_1, x_{18}), (\tau_2, (\tau_1, x_{17}), (\tau_1, x_{15}))), (\lambda x_{16}. \lambda x_{17}. (\tau_2, (\tau_1, x_{16}), (\tau_2, (\tau_1, x_{17}), (\tau_1, x_{18}))), (\lambda x_{10}. (\tau_1, x_{10}))) by 5
\rightarrow \lambda x_{18}. (\lambda x_{16}. \lambda x_{17}. (x_{16}(x_{17}, x_{18}))(\lambda x_{10}. x_{10}))
\end{align*}
\]

This is equivalent to the original expression up to \(\alpha\)-conversion if

\[
\begin{align*}
f &= x_{18} \\
x &= x_{16} \\
y &= x_{17} \\
p &= x_{10}
\end{align*}
\]
An Algebra for MDSC

In this section, we will define an algebra which can be uniformly applied to any \(\lambda\)-abstraction. This system simplifies the work done by Sreedhar and Taghva [29] because it eliminates the need for case-by-case analysis.

To assist in defining the algebra, let \(A\) and \(B\) be director strings of any length, \(D1\) and \(D2\) be single director strings, and \(<>\) be the empty director. The following operations are defined on directors and director strings:

**Disjunction (\(+\)**)

\[
\# + \# = \#
\]
\[
# + ! = !
\]
\[
! + \# = !
\]
\[
!+! = !
\]
\[
<> + B = B
\]
\[
A + <> = A
\]
\[
(D1.A) + (D2.B) = (D1 + D2). (A + B)
\]

Therefore, \(A + B\) transmits everything that either \(A\) or \(B\) transmits.

**Conjunction (\(*\)**)

\[
\# * \# = \#
\]
\[
# *! = \#
\]
\[
! * \# = \#
\]
\[
!*! = !
\]
\[
<> * B = B
\]
\[
A * <> = A
\]
\[
(D1.A) * (D2.B) = (D1 * D2). (A * B)
\]
Therefore, $A \ast B$ transmits only things that both $A$ and $B$ transmit.

Sequential Composition (;)

$<>; B = B$

$A; <> = A$

$(#A; D1.B) = #.A; B$

$(!A; D1.B) = D1.(A; B)$

This describes the transmission of arguments. If $x$, $y$ and $z$ are nodes such that $x$ is the parent of $y$ and $y$ is the parent of $z$, and $A$ is the director string transmitting incoming arguments from $x$ to $y$, and $B$ transmits arguments from $y$ to $z$, then $A; B$ is a string specifying which incoming arguments to $x$ reach $z$.

As with calculus presented in [29], we use the shift, insert, and remove functions. We will also introduce a new function fail.

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

- **insert**$(E, i, j)$ is also defined to be a term obtained from $E$ by inserting $j$ numbers of # between $(i + 1)$ and $(i + 2)$ positions of each director string $E$.

- **remove**$(E, i)$ is defined to be a term obtained from $E$ by removing $(i + 1)$st director of every director in $E$.

- **fail**$(n)$ is defined to be a string of discards of length $n$.

For example:

$shift(#!# : #!#, 1, 1) = #!# : #!#$

$remove(#!# : #!#, 1) = #! : #!$
The insert function is used to avoid variable clashes by performing \( \alpha \)-conversion, shift is used to preserve the relationship between a director and its binding, and remove is used to indicate that a \( \beta \)-reduction has been done; this implies that a particular director will no longer be needed.

Let a general application node have the representation shown in figure 4.6 where \( L_0, R_0, L_1, R_1, E_1, E_2, \) and \( F \in MDST \). A redex exists when the length of \( L_0 \) is less than the length of \( L_1 \) (\(| L_0 | < | L_1 | \)). When a redex is detected at \( L_0 \), the director strings \( L_1 \) and \( R_1 \) are divided as follows:

\[
L_1 = L_1'.LD.L_1''
\]
\[
R_1 = R_1'.RD.R_1''
\]

where \(| L_1' | = | R_1' | = | L_0 | \)

\( LD \) and \( RD \) are single directors

\( L_1'' \) and \( R_1'' \) are director strings of length zero or more.

The division of \( L_1 \) and \( R_1 \) in this manner allows us to get to the first variable in the expression which is not affected by any director symbol in \( L_0 \). The single directors \( LD \) and \( RD \) bind the variable that we are currently processing, (say \( x \)); they also
dictate which binary rule is used. For instance, if $LD = \lambda$, then $x$ occurs free in expression $E_1$ and we should perform a $\beta$-reduction replacing free occurrences of $x$ in $E_1$ by $F$. $F$ is the argument to the function in the same way that $G$ was the argument to application $(EF)$ in figure 4.1. If $RD = \#$, then $x$ is not free in $E_2$, so no substitution occurs during $\beta$-reduction, and the variable $x$ is no longer needed.

The binary reduction rules are defined in figures 4.7, 4.8, 4.9, and 4.10.
When a redex is detected, we apply various operations of the algebra as shown in figure 4.11. The operations are described as follows:

- \( L0' = ((L0; L1') + R0).L1" \)
  
  \( L0' \) passes variables to the left. It must pass any variables that are used by either \( E1 \) or \( F \). The variables passed to \( E1 \) from the original redex at \( L0 \) are defined by \( (L0; L1') \). The addition \((+\) of \( R0 \) includes variables that were originally sent to \( F \). Finally, the concatenation of \( L1" \) includes variables that were sent to \( E1 \) from the application node below the redex.

- \( R0' = ((L0; R1') + R0).R1" \)
  
  Controls variables in a way similar to \( L0' \).

- \( L2' = (L0; L1').L1" \)
  
  This director must pass any variables that were used by \( E1 \). \( (L0; L1') \) includes the variables that were passed to \( E1 \) from the original redex at \( L0 \). The concatenation of \( L1" \) includes variables that were sent to \( E1 \) from the application node below the redex.

- \( R2' = R0.FAIL(|L2'| - |R0|) \)
  
  This director must pass any variables that were used by \( F \); by definition, this is \( R0 \). The concatenation of discards using the \( FAIL \) operator is needed to ensure that \(|L2'| = |R2'| \)

- \( L3' \) and \( R3' \) are similar to \( L2' \) and \( R2' \)
The expressions $E_1'$, $E_2'$ and $F'$ are obtained by rearranging the director strings occurring in $E_1$, $E_2$ and $F$ using shift, insert and remove. Upon application of $(E_1 \ E_2)$ to $F$, the binary rules produce applications of the form $(E_1 \ F)$ and $(E_2 \ F)$. This could lead to the situation depicted in figure 4.12. Again, a redex exists when $|L_0| < |L_1|$. When a redex is detected, $L_1$ is written as $L_1 = L_1'.L.D.L_1''$, where with $|L_1'| = |L_0|$ and $|L.D| = 1$. This creates two cases depending on whether $L.D$ is $!$ or $\#$ and produces two unary rules.

Unary Rule II.1

$$(L_0 \ R_0, (L_1'.L_1'', E), (R_1, F)) \rightarrow (R_1, F)$$

Unary Rule II.2

$$(L_0 \ R_0, (L_1'.\#L_1'', E), (R_1, F)) \rightarrow (L_1'.L_1'', E)$$
The two rules are defined in figures 4.13 and 4.14. If we have a unary node without a redex ($|L0| > |E|$), then it must be 'collapsed' using the conjunction operator as shown in figure 4.15.

The following examples will demonstrate the correctness of these rules. First, begin with $\lambda x.(\lambda y.(xy))x$ which was used to show that Kennaway’s Director String Calculus did not capture strong reduction. The reduction sequence is shown in figure 4.16. Beginning with the redex in the first tree marked with the $\ast$, and following binary rules I.1 through I.4 as well as the calculations shown in figure 4.11,

$$LO = ! \Rightarrow |L0| = 1$$
Where \( L0' = L0 \times E1 \)
\( R0' = R0 \times F \)

Figure 4.15: A general collapsed application node.
Figure 4.16: $\lambda x.(\lambda y.(xy))x$
\[ R_0 = ! \Rightarrow |R_0| = 1 \]

\[ L_1 = !\# \Rightarrow L_1' = !, \quad LD = \# \]

\[ R_1 = \#! \Rightarrow R_1' = \#, \quad RD = ! \]

\[ LD = \#, \quad RD = ! \Rightarrow 1.3 \]

Referring to figures 4.9 and 4.11, we use the following rules:

\[ L0' = ((L0; L1') + R0).L1'' \]
\[ = (\!(\!\!\!)\!+)\!\!\!.<> \]
\[ = ! \]

\[ R0' = ((L0; R1') + R0).R1'' \]
\[ = (\!(\!\!\!\#\!)\!+)\!\!\!.<> \]
\[ = ! \]

\[ L3' = (L0; R1').R1'' \]
\[ = (\!(\!\!\!)\!\#\!).<> \]
\[ = \# \]

\[ R3' = R0.FA I L( |L3' | - |L0' | ) \]
\[ = \!.FA I L( 1 - 1 ) \]
\[ = ! \]

\[ E1' = remove( E1, |L1'| ) \]
\[ = remove( \!\#\!, 1) \]
\[ E_2' = \text{shift}(E_2, |L_1'|, |L_1''|) \]
\[ = \text{shift}(\#, 1, 0) \]
\[ = \#! \]

\[ F' = \text{insert}(F, |L_1'|, |L_1''|) \]
\[ = \text{insert}(!, 1, 0) \]
\[ = ! \]

After performing this sequence of operations, we can move to the next tree in the figure which is a unary node.

\[ L_0 = \# \Rightarrow |L_0| = 1 \]
\[ E_1 = \#! \Rightarrow L_1' = \# \text{ and } LD =! \]

\[ LD =! \Rightarrow \text{Rule II.1, insert } (!, \Delta) \]

This leads us to the third tree in figure 4.16. No redex exists, but we have a unary node which must be collapsed using conjunction. The operation is:

\[ \text{!}*! =! \]

which results in the last tree in figure 4.16, and this is \( \lambda x. xx \) as desired.

The next example will use the expression \( \lambda f.(\lambda x.\lambda y.x(yf))\lambda p.p \) from figure 4.5; the reduction sequence is shown in figure 4.17. As before, begin with the redex at \(*\),
Figure 4.17: Reduction of $\lambda f. (\lambda x. \lambda y. x(yf)) \lambda p. p$

$L0 = \! \Rightarrow |L0| = 1$

$R0 = \# \Rightarrow |R0| = 1$

$L1 = \#!\# \Rightarrow L1' = \#, L1'' = \#$

$R1 = !\#! \Rightarrow R1' = !, R1'' = !$

$LD = !, RD = \# \Rightarrow I.2$

Referring to figures 4.8 and 4.11, we use the following rules:

$L0' = ((L0; L1') + R0).L1''$

$= ((!; \#) + \#).\#$

$= \#\#$

$R0' = ((L0; R1') + R0).R1''$

$= ((!; !) + \#).!$

$= !!$
\[ L_2' = (L_0; L_1').L_1'' \]
\[ = (\#; \#).\# \]
\[ = \#\# \]

\[ R_2' = R_0.FA Il( |L_2'| - |R_0'| ) \]
\[ = R_0.FA Il( 2 - 1 ) \]
\[ = \#\# \]

\[ E_1' = shift( E_1, |L_1'|, |L_1''| ) \]
\[ = shift( \#!\#, 1, 1 ) \]
\[ = \#\#! \]

\[ E_2' = remove( E_2, |L_1'| ) \]
\[ = remove( \#!\# !\#\# , 1 ) \]
\[ = \#! !\# \]

\[ F' = insert( F, |L_1'|, |L_1''| ) \]
\[ = insert( \#!, 1, 1 ) \]
\[ = \#\#! \]

Moving to the next redex *, we have a unary redex using rule II.1.
\[ L_0 = \#\# \Rightarrow |L_0| = 2 \]
\[ E_1 = \#\#! \Rightarrow L_1' = \#\# \text{ and } LD =! \]

\[ LD =! \Rightarrow \text{Rule II.1, insert } (\#\#, \Delta) \]

Finally, moving to the next to the last tree, again we have a unary redex using rule II.1.

\[ L_0 = \#\# \Rightarrow |L_0| = 2 \]
\[ E_1 = \#\#! \Rightarrow L_1' = \#\# \text{ and } LD =! \]

\[ LD =! \Rightarrow \text{Rule II.1, insert } ((#!, \Delta), (!#, \Delta)) \]

This represents \( \lambda f.\lambda y.(yf) \) up to \( \alpha \)-conversion as desired.

Appendix A shows a more detailed example based on Church’s Numerals.

**Conclusion and Future Work**

We have given a simple and mathematically more elegant algorithm for strong director string calculus as defined in [7]. This should make future work in the area easier to explain and pursue.

A significant area of future work is to determine the size of the transformations used to capture strong reduction. It is clear that a simple case like Church’s Numeral gets large quite quickly. Further work is needed to determine if the algebra is of enough benefit to justify the implementation cost.

There are similarities between the MDSC and de Bruijn \( \lambda \)-calculus. The position of the director is related to the de Bruijn number. The relative position of a variable is more important than the variable name in both de Bruijn calculus and MDSC; each system uses this position to avoid \( \alpha \)-conversion. Exploration into the similarities of these two systems may prove worthwhile.
Appendix A

Church's Numeral

The appendix will present the following application of Church's Numeral.

\[(\lambda x.\lambda y.(xxy))(\lambda x.\lambda y.(xxy))\] by \(\lambda\)-calculus \(\beta\)-reduction the normal form of this expression is \(\lambda x.\lambda y.(x(x(xy)))\)
No redexes remain, and the resulting expression is "4" as desired.
Bibliography


