INFORMATION TO USERS

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

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 copyrighted 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 oversize page is available as one exposure on a standard 35 mm slide or as a 17” × 23” black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. 35 mm slides or 6” × 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.
Efficient list manipulation in combinator-based functional languages on parallel architectures

Sarwar, Syed Mansoor, Ph.D.

Iowa State University, 1988
Efficient list manipulation in combinator-based functional languages on parallel architectures

by

Syed Mansoor Sarwar

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY
Department: Electrical Engineering and Computer Engineering
Major: Computer Engineering

Approved:

Signature was redacted for privacy.  
In Charge of Major Work
Signature was redacted for privacy.  
For the Major Department
Signature was redacted for privacy.  
For the Graduate College

Iowa State University
Ames, Iowa
1988
TABLE OF CONTENTS

1 INTRODUCTION ....................................................... 1
  1.1 Introduction .................................................... 1
  1.2 Problem Statement .............................................. 4
  1.3 Thesis Organization ............................................. 5

2 BACKGROUND MATERIAL AND PROBLEM DESCRIPTION 6
  2.1 Introduction .................................................... 6
  2.2 Abstraction ..................................................... 8
    2.2.1 Turner's abstraction method ................................ 9
    2.2.2 Abdali's abstraction method ................................ 16
    2.2.3 Hughes' abstraction method ................................ 20
  2.3 Detailed Problem Description ................................... 24

3 TERMINOLOGY, NOTATION, AND PRELIMINARY COMBINATORS ............ 31
  3.1 Introduction .................................................... 31
  3.2 Terminology and Notations ...................................... 31
  3.3 Preliminary Combinators ........................................ 36

4 COMBINATORS FOR LIST MANIPULATION .............................. 40
# List Combinators

4.1 Introduction ................................................. 40

4.2 Basic List Combinators ........................................ 42

4.3 Major List-Manipulation Combinators .......................... 46

4.3.1 List indexing .............................................. 47

4.3.2 Taking elements from the head of a list ................... 48

4.3.3 Taking elements from the tail of a list .................... 49

4.3.4 List expansion (insertion) ............................... 49

4.3.5 Deletion of list elements .................................. 51

4.3.6 Modifying the list elements ............................... 52

4.3.7 Conversion of a list to multiple sublists ................. 53

4.3.8 Transpose of a list of lists ............................. 54

4.3.9 Reversing a list .......................................... 57

4.4 Parallel List Combinators .................................... 58

4.4.1 distl (DL) .................................................. 59

4.4.2 distr (DR) .................................................. 60

4.4.3 Inner Product (IP) ......................................... 61

4.4.4 MAP ........................................................ 62

4.5 An Illustrated Example ....................................... 63

5 LIST REPRESENTATION ......................................... 66

5.1 Introduction .................................................. 66

5.2 The Concurrent List .......................................... 66

5.2.1 Logical representation of the concurrent list ........... 67

5.2.2 Memory management ....................................... 69

5.3 Examples ...................................................... 70


LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.1</td>
<td>SASL script and combinatory codes for matrix multiply</td>
<td>25</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>First 33 reduction steps for map (plus) (1,2) using Turner's combinatory code</td>
<td>27</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>First 13 reduction steps for map (plus 1) (1,2) using Abdali's combinatory code</td>
<td>28</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>Concurrent list representation of (1,2, ...,100)</td>
<td>72</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>Concurrent list representation of (100,99, ...,1)</td>
<td>74</td>
</tr>
<tr>
<td>Figure 5.3</td>
<td>Concurrent list representation after the insertion of 3,4, and 5 at the 17th position</td>
<td>76</td>
</tr>
<tr>
<td>Figure 5.4</td>
<td>Concurrent list representation after the deletion of 7, ...,13 from the list</td>
<td>78</td>
</tr>
<tr>
<td>Figure 6.1</td>
<td>Correspondence between the matrix size and the reduction steps for matrix multiply</td>
<td>80</td>
</tr>
</tbody>
</table>
1 INTRODUCTION

1.1 Introduction

In recent years, interest in functional languages [2,8,13], their implementation, and architectural support has increased rapidly. A number of researchers have advocated the use of functional languages as a means for solving the software crisis in terms of increasing programmer productivity, enhancing program reusability, improving clarity of programs, and easing program verification. Although their elegance has long been recognized, functional languages have not proven competitive because of their slow execution on von Neumann machines, for the von Neumann architecture does not support the underlying execution model of these languages.

A conventional method of implementing functional languages is to use the environment model [12]. The environment is a function with a domain of variable names currently in scope and a range of values associated with each of these variables. This model uses a data structure to store these name-value pairs. Two major costs involved in the environment model are the construction of closures and searching the environment sequentially to evaluate an expression. The program execution time can be decreased enormously if the construction of closures can be avoided and the environment is distributed so that only a portion needs to be searched to evaluate an expression.
A radically different method of implementing functional languages is the combinator-based reduction model. This model was first proposed by Turner [16] using a result in mathematical logic which says that variables as they are used in ordinary mathematics are not strictly necessary because they are referentially transparent. Using this method, functional languages can be translated into a form that does not have any bound variables. The place of variables is taken by primitive operators, called combinators [6], and a mechanism for applying a function to its arguments. The consequence of having a variable-free form of a program is that there is no need to maintain an environment, and the program can be executed by a simple-natured machine whose instruction set is comprised of reduction rules for the combinators. The operation of this machine consists of progressively transforming the compiled (combinatory) code into a form which can be printed directly. Such a machine is also known as a substitution machine because it carries out literal substitutions on the combinatory code; a combinator is replaced by the right hand side of its reduction rule, and a free variable (a user-defined or a library function) is replaced by its definition elsewhere in the language run-time system.

Programs in a substitution machine evince two primary self-optimizing properties [16]. Both of these optimizations are a consequence of the semantics of the reduction model and not the combinatory nature of the executable program. First, constant calculations are replaced by their values and, therefore, moved outside the loop (if it so happens to be the case) and evaluated only once. A second advantage is achieved when a programming style is used in which extra levels of functional abstraction are introduced into the programs. In other words, a typical function, instead of being defined directly by a step-by-step sequencing of instructions, is
expressed as an application of a general-purpose function to its arguments. Such a definition is called the combinatory definition of a function [3]. The arguments of the generic function could be previously defined functions which may themselves have combinatory definitions. Under an environment model based implementation, such a technique causes a substantial slow-down in program execution after a certain level of abstraction. In the reduction model, however, only the first call to a function has overhead associated with it; cost for all subsequent calls is mitigated because of the substitution nature of the execution model.

The combinator-based substitution model has a major disadvantage, namely the combinatory explosion. This is particularly dangerous when a set of fixed combinators is used and a recursive abstraction mechanism is employed. This combinatory explosion can be avoided by either employing an abstraction mechanism that abstracts all variables in one step [1], or using the lambda-lifting [13] technique to create a (non-fixed) set of super-combinators [11]. The first technique results in a machine with a complicated instruction set and the second mechanism saves a number of full or partial expressions, because the super-combinators do not have any fixed reduction rules and are defined and created at compile time. Although these and some other techniques (e.g., serial-combinators [9,10] and lazy super-combinators [14]) make functional programs run faster, they do not address carrying out abstraction on elements of a list. Therefore, although the parallelism available at the source program level becomes available in a number of steps equal to the number of variables in the program, the evaluation of functional subexpressions that involve list manipulation remains recursive (i.e., sequential).

The recursive list representation and the lack of list combinators force list
manipulation to be performed sequentially, using primitive operators like CONS, CAR, and CDR. The author calls the recursive definition of the list structure and its sequential and recursive manipulation the *functional bottleneck*. It seems as if we have taken away the assignment statement in an effort to remove the *von Neumann bottleneck* but have overlooked the recursive and sequential nature of list representation and manipulation. Therefore, although referential transparency saves reevaluation of expressions within a scope, the von Neumann bottleneck still manifests its behavior at the architecture level by manipulating list one element at a time.

### 1.2 Problem Statement

The work described in this dissertation is an attempt to overcome the functional bottleneck in a combinator-based reduction environment. To accomplish this, a set of combinator families and a new list representation that maps well onto the semantics of these combinators are introduced to manipulate list elements concurrently. Most of these combinators reshape the structure of the list so that higher-order combinators can perform concurrent operations on elements of the resultant list. The reduction semantics and proofs of correctness for these combinators are given in the form of combinatory strings of already known combinators. These basic list combinators are then used to define higher-level combinators that perform sequential and parallel operations on list elements very efficiently. It will be shown that these combinators and the proposed list structure would make the execution of functional programs very fast on both sequential and parallel machines.
1.3 Thesis Organization

The remainder of this dissertation describes the proposed list combinators and list representation to remove the functional bottleneck.

In Chapter 2, the background material pertaining to the lambda calculus and combinators is reviewed and the problem description is explained in more detail by way of an example.

In Chapter 3, terminology, notation, and some preliminary combinators are introduced. The notations and combinators described in this chapter act as the basis for the list combinators described in Chapter 4.

In Chapter 4, three levels of list combinator families are introduced. The first level combinators are used to construct and destroy lists in an efficient manner. The second level combinators (based upon the first-level list combinators and the preliminary combinators) are developed to manipulate the list elements in a number of ways and encompass most of the list operations. Finally, a set of list combinator families is introduced to perform map-like parallel operations on list.

In Chapter 5, a new list representation is proposed that very nicely supports the semantics of the proposed combinators. Some examples are given to show the mapping of some of the combinators onto this list structure.

Finally, in Chapter 6, the set of proposed combinators are evaluated and an outline is given for future research.
2 BACKGROUND MATERIAL AND PROBLEM DESCRIPTION

2.1 Introduction

In this chapter, we describe three methods of translating a functional program into a form which has no variables and consists entirely of a set of constant definitions. These constants, called combinators, along with a mechanism of applying a function to its arguments, express the role of abstracted variables. The combinatory code is then executed on a substitution machine where each combinator becomes a machine instruction. Before describing the abstraction algorithms, however, we review some of the fundamental characteristics of \lambda-calculus [5] and combinators.

\lambda-calculus was proposed by Alanzo Church in the 1930s as a collection of formal systems to describe functions and function application by using a special notation. Church's notation is a systematic way of constructing, for each expression involving \( x \), a notation for the corresponding function of \( x \), for each expression involving \( y \), a notation for the corresponding function of \( y \), etc. For example, if there are two functions \( f(x) = x + y \) and \( f(y) = x + y \), in \lambda-notation they can be written as \( \lambda x.x + y \) and \( \lambda y.x + y \), respectively. The terms involving \lambda's are called \( \lambda \)-terms or \( \lambda \)-abstractions. A \lambda-abstraction is composed of two parts: a bound variable part and a body. In the above example, \( \lambda x \) and \( \lambda y \) are the bound variable parts (\( x \) and \( y \) being the bound variables) and \( x + y \) is the body. For completeness, \( y \) and \( x \) are free variables.
in the first and the second λ-abstractions, respectively. Formally, a λ-term or a λ-expression is defined inductively as:

1. Every variable and constant is a λ-expression.

2. If M and N are any λ-expressions, then \((MN)\) is a λ-expression (called an application).

3. If M is any λ-expression and x is any variable, then \((\lambda x.M)\) is a λ-expression (called an abstraction).

Following are some examples of λ-expressions:

\[
\begin{align*}
3 \\
x \\
(\lambda x.x) \\
(\lambda x.xy) \\
(((\lambda x.x)(\lambda x.xy)) \\
(x(\lambda x.(\lambda x.x))) \\
(\lambda x.yz)
\end{align*}
\]

A λ-expression of the form \((\lambda x.M)N\) represents a function \(\lambda x.M\) applied to argument \(N\) and is called a β-redex or a redex (for reducible expression). This application results in a λ-expression \(M'\) which is the λ-expression \(M\) with \(N\) having been substituted for every occurrence of the bound variable \(x\) in its body. The step of substituting an argument for a bound variable is called a β-reduction or a β-contraction or a reduction. An expression that does not contain a β-redex is said to be in β-normal form or normal form. For an example, consider the λ-expression
(\lambda z. x y) \beta 3. This is a \beta-redex and one \beta-reduction results in 3y, which is also the normal form for the given \lambda-expression.

By definition, a combinator is a \lambda-expression that has no free (global) variables and functional application is the only operation used in the expression. Intuitively, however, a combinator is a place-holder for abstracted variable(s), and the reduction rule of a combinator dictates where actual parameters are substituted. Current interest in combinators arose out of the desire to generate variable-free code for a functional program. The significance of not having variables in the code is that there is no need for constructing closures and maintaining an environment. It is well-established that the construction of closures, maintenance of an environment, and subsequent name look-up is extremely time-expensive in traditional implementations of von Neumann languages. Environment-based lazy implementations of functional languages have also shown poor performance compared to the combinator-based lazy implementations, both in terms of code size and speed of execution [16].

2.2 Abstraction

The basis for any abstraction scheme is the use of higher-order functions, the principle of extensionality, and the law of abstraction. The higher-order functions are functions that take functions as input and/or return functions, instead of simple values, as output. We normally write these functions in their curried format. The following are examples of such functions:

1. The differentiate function, D. For example, application of D to a \sin function returns another function, \cos.
2. The plus function, PLUS. For example, PLUS x y = (PLUS x) y, where (PLUS x) is a function, which when applied to y returns (x PLUS y).

The principle of extensionality says that two functions, say f and g, are equal if and only if \( f x = g x \) for all x. It is because of this principle that we can abstract variables from a functional expression. For example, if \( suc x = plus 1 x \) then due to the principle of extensionality we can say \( suc = plus 1 \).

The law of abstraction says that abstraction is an exact inverse of application. We denote abstraction of a variable x from a functional expression E by \( ([x] E) \), which stands for a term containing no occurrences of x. According to the law of abstraction then, \( ([x] E) x = E \). Thus, \( ([x] (f x)) x = f x \).

There are three well-known abstraction methods, proposed by Abdali [1], Turner [16], and Hughes [11]. Where Turner's and Abdali's methods are based upon the same basic principle, Hughes uses an altogether different methodology. Each method is briefly described in the following discussion.

2.2.1 Turner's abstraction method

Turner proposes a compilation technique for functional languages that eliminates bound variables in a program and produces a combinatory code. His method abstracts one instance of a variable at a time from a functional expression. If a functional expression is denoted by \( E \) and a variable by x, then the abstraction notation is:

\[ [x] E \]

Turner's basic abstraction algorithm performs the following two steps on a function body:
1. Replaces all the operators by their curried versions. (Currying is the process of reducing a first order function of several arguments to a higher-order function of one argument.)

2. Removes (by abstraction) all the variables in the resulting function body an instance of a variable at a time according to the following abstraction rules:

**Rule 1:** If $E$ is a variable or a constant that exactly matches $x$, then

$$[x] E \rightarrow I$$

Examples:

$$[x] x \rightarrow I$$
$$[y] y \rightarrow I$$

**Rule 2:** If $E$ is a variable or a constant other than $x$, then

$$[x] E \rightarrow K E$$

Examples:

$$[x] y \rightarrow K y$$
$$[x] 3 \rightarrow K 3$$

**Rule 3:** If $E$ can be written as $(E_1 E_2)$, then

$$[x] E \rightarrow S ([x] E_1) ([x] E_2)$$

Examples:

$$[x] + 1 x \rightarrow S ([x] + 1) ([x] x)$$
$$[n] * n fac (- n 1) \rightarrow S ([n] * n fac) ([n] - n 1)$$
Here, I, K, and S are combinators with the following reduction rules:

\[ I f \rightarrow f \]
\[ K f g \rightarrow f \]
\[ S f g h \rightarrow f h (g h) \]

Turner also defines some optimization rules (specifically, the B and C combinators) to avoid the space explosion of the combinatory code that is generated by using his abstraction algorithm. Following are his optimization rules (\(E_1\) and \(E_2\) stand for arbitrary combinations):

\[ S (K E_1)(K E_2) \rightarrow K (E_1 E_2) \]
\[ S (K E_1) I \rightarrow E_1 \]
\[ S (K E_1) E_2 \rightarrow B E_1 E_2 \]
\[ S E_1 (K E_2) \rightarrow C E_1 E_2 \]

The reduction rules for B and C combinators are:

\[ B f g h \rightarrow f (g h) \]
\[ C f g h \rightarrow f h g \]

Finally, Turner introduces B1, C1, and S1 combinators [17] to avoid abstractions on the longest initial component composed of combinators to make the combinatory code still more compact. The purpose of these combinators is to avoid combinatory code explosion when the abstraction algorithm is applied to an expression of more than one variable. With the use of these combinators, the size of the combinatory code increases linearly with the number of variables abstracted, as opposed to quadratically without the use of these combinators. The reduction rules
for these combinators are:

\[ B1 \ k \ f \ g \ h \ \rightarrow \ k(f(g(h))) \]
\[ C1 \ k \ f \ g \ h \ \rightarrow \ k(f(h)g) \]
\[ S1 \ k \ f \ g \ h \ \rightarrow \ k(f(h)(f(h)) \]

After having translated a program into a combinatory code, Turner builds a graph from this code and the actual parameters of the program. Turner then reduces this graph by searching it inorder for a redex. Once a redex is found, the graph is reshaped by performing a reduction, and the search continues for the next redex until no more redex is available for reduction. The resultant graph is then said to be in normal form. In the discussion to follow, however, we will illustrate different aspects of the combinatory-graph reduction by way of string reduction.

Now, we analyze a simple SASL [18] program to illustrate the above-mentioned abstraction, optimization, and reduction processes. Consider the following function definition:

\[ \text{DEF} \]
\[ f \ x \ = \ x \ ** \ 2 \ - \ 1 \]

where "**" denotes exponentiation.

The complete abstraction process for this function is given below:

- Step1 of the algorithm gives MINUS (POWER x 2) 1

- Step2 of the algorithm results in the following sequence of abstraction operations:
\[ x \] \text{MINUS} (\text{POWER} \times 2) 1
\rightarrow S ([x] \text{MINUS} (\text{POWER} \times 2)) ([x] 1)
\rightarrow S (S ([x] \text{MINUS}) ([x] \text{POWER} \times 2)) ([x] 1)
\rightarrow S (S (K \text{MINUS}) ([x] \text{POWER} \times 2)) ([x] 1)
\rightarrow S (S (K \text{MINUS}) (S ([x] \text{POWER} \times 2)) ([x] 1))
\rightarrow S (S (K \text{MINUS}) (S (S ([x] \text{POWER} \times 2)) ([x] 2)) ([x] 1))
\rightarrow S (S (K \text{MINUS}) (S (S (K \text{POWER}) ([x] 2)) ([x] 1))
\rightarrow S (S (K \text{MINUS}) (S (S (K \text{POWER}) (K 2)) ([x] 1))
\rightarrow S (S (K \text{MINUS}) (S (S (K \text{POWER}) (K 2)) (K 1)) (K 1)

Note that this form of the program does not contain any variables and consists entirely of a number of constants (combinators, operators, and integers). The application of the optimization rules results in the following steps:

\[ S (S (K \text{MINUS}) (S (S (K \text{POWER}) (K 2)) (K 1)) \rightarrow S (S (K \text{MINUS}) (S \text{POWER} (K 2)) (K 1)
\rightarrow S (S (K \text{MINUS}) (C \text{POWER} 2)) (K 1)
\rightarrow S (B \text{MINUS} (C \text{POWER} 2)) (K 1)

Note that the size of this combinatory code is about 50% of the size of the original code. If B1, C1, and S1 combinators are used, we obtain \textit{C1 MINUS (C POWER 2)} 1 as the combinatory code, which is a size reduction of about 70% over the original code. If an expression contains more than one variable, the reduction in the code size will be larger because B1, C1, and S1 combinators are designed to generate efficient code for multi-variable expressions.
We complete this example by showing how the combinatory code is evaluated. The execution of the function $f$ for an argument of 3 results in the following reduction steps:

$$S (B \text{MINUS} (C \text{POWER} 2)) (K \ 1\ 3)$$

$$\equiv B \text{MINUS} (C \text{POWER} 2)\ 3\ (K \ 1\ 3)$$

$$\equiv \text{MINUS} (C \text{POWER} 2\ 3)\ (K \ 1\ 3)$$

$$\equiv \text{MINUS} \ (\text{POWER} 3\ 2)\ (K \ 1\ 3)$$

$$\equiv \text{MINUS} \ 9\ (K \ 1\ 3)$$

$$\equiv \text{MINUS} \ 9\ 1$$

$$\equiv 8$$

The answer is in normal form because 8 does not contain a reducible expression.

Some of the major disadvantages of Turner's abstraction algorithm are:

1. The compilation is slow because:

   (a) Optimization rules are applied after having translated the source program into a combinatory code.

   (b) Because the algorithm abstracts one instance of one variable at a time, the source code must be scanned many times.

2. The translation is expensive. For a program size of $N$, the average and the worst-case complexities are $O(N \ logN)$ and $O(N^2)$, respectively.

3. During the reduction of the combinatory code, substitution of the actual parameters takes place one at a time, thereby making the reduction process sequential and slow.
4. The combinatory code (which is actually the machine code for the underlying reduction machine) is far removed from the source program thereby making human interpretation of the intermediate values difficult during debugging.

5. Execution is broken down into very small granules, making the overhead of linking one step to the next fairly high. In other words, the amount of work done by a combinator is much smaller than the amount of stack manipulation needed to perform a reduction and get the run-time system ready for the next reduction step.

6. Since the actual arguments to a function are pushed down into its body one level at a time, many intermediate nodes are created and almost immediately taken apart again. This means that the reduction machine consumes a lot of transient storage, which increases the load on the garbage collector.

Despite these drawbacks, Turner's abstraction method has a number of advantages:

1. The method is simple and elegant.

2. The method results in very simple instruction set for the underlying architecture.

3. The combinators generated by this method are very simple and can be directly implemented in hardware.

4. The technique is fully lazy. That is, during the reduction process, an expression is evaluated (reduced) only if needed and is evaluated only once throughout the execution of the program.
2.2.2 Abdali's abstraction method

Abdali [1] proposes an abstraction method that performs abstraction on multiple variables at a time. If a functional expression is denoted by $E$ and is composed of variables $X_1, X_2, \ldots, X_n$, then the abstraction notation is:

$$[X_1, X_2, \ldots, X_n] E$$

Abdali's abstraction algorithm performs the following two steps on a function body:

1. Replaces all the operators by their curried versions.

2. Abstracts all the variables in the resulting function in one step according to the following rules:

(A) If $X_i$ is not in $E$ for all $i$, $1 \leq i \leq n$, then

$$[X_1, X_2, \ldots, X_n] E \rightarrow K_n E$$

Examples:

$$[x] a \rightarrow K_1 a$$

$$[x,y] 2 \rightarrow K_2 2$$

$$[x,y] + a b \rightarrow K_2 + a b$$

(B) If $E$ matches $X_1 X_2 \ldots X_n$, then

$$[X_1, X_2, \ldots, X_n] E \rightarrow I$$

Examples:

$$[x] x \rightarrow I$$

$$[x,y] x y \rightarrow I$$

$$[x,y] x y a b \rightarrow I a b$$
(C) If $E$ matches $X_i$ for some $i$, $1 \leq i \leq n$, then

$$[X_1, \ldots, X_n] E \rightarrow I_n^i$$

Examples:

- $[x] x \rightarrow I_1^1$
- $[x, y] y \rightarrow I_2^2$
- $[x, y, z] x \rightarrow I_3^1$

(D) If $E$ is of the form $g X_1 \ldots X_n$ and $X_i$ is not contained in $g$ for all $i$, $1 \leq i \leq n$, then

$$[X_1, \ldots, X_n] E \rightarrow g$$

Examples:

- $[x] + x \rightarrow +$
- $[x] + 3 x \rightarrow + 3$
- $[x, y] + x y \rightarrow +$

(E) If $E$ is of the form $g X_m \ldots X_n$ and $X_i$ is not contained in $g$ for $i \geq m$, then

$$[X_1, \ldots, X_n] E \rightarrow [X_1, \ldots, X_{m-1}] g$$

Examples:

- $[x, y] + y \rightarrow [x] +$
- $[x, y, z] + + 1 w z \rightarrow [x, y] + + 1 w$
- $[x, y, z] + + + 1 w y z \rightarrow [x] + + + 1 w$

(F) If $E$ is of the form $X_i f_2 \ldots f_m$ for some $i$, $1 \leq i \leq n$, then

$$[X_1, \ldots, X_n] E \rightarrow B_n^m I_n^i ([X_1, \ldots, X_n] f_2) \ldots ([X_1, \ldots, X_n] f_m)$$
Examples:

\[
[x] x 1 \rightarrow B^2_1 I I^1_1 ([x] 1)
\]

\[
[x] x a b c \rightarrow B^4_1 I I^1_1 ([x] a) ([x] b) ([x] c)
\]

\[
[x, y] y a b c \rightarrow B^4_2 I I^2_2 ([x, y] a) ([x, y] b) ([x, y] c)
\]

\((G)\) If \(E\) is of the form \(f_1 \ldots f_m\) where \(f_1\) is the longest initial component not containing an \(X_i\) for all \(i, 1 \leq i \leq n\), then

\[
[X_1, \ldots, X_n] E \rightarrow B^{m-1}_n f_1 ([X_1, \ldots, X_n] f_2) \ldots ([X_1, \ldots, X_n] f_m)
\]

Examples:

\[
[x] a x \rightarrow B^1_1 a ([x] x)
\]

\[
[x, y] a b (c y) (d x) \rightarrow B^2_2 ([x, y] (c y)) ([x, y] (d x))
\]

\[
[x, y] POWER y 2 \rightarrow B^2_2 POWER ([x, y] y) ([x, y] 2)
\]

The reduction rules for the combinators that are generated as a result of above abstraction rules are the following:

\[
K_n a b_1 \ldots b_n \rightarrow a
\]

\[
I^m_n a_1 \ldots a_n \rightarrow a_m
\]

\[
B^m_n a b_1 \ldots b_m c_1 \ldots c_n \rightarrow a (b_1 c_1 \ldots c_n) \ldots (b_m c_1 \ldots c_n)
\]

As an example, the function definition that was used to illustrate the abstraction, optimization, and reduction processes for Turner's algorithm is used to illustrate Abdali's abstraction method and reduction rules for his combinators. The application of the algorithm on the function body consists of the following sequence
of abstraction steps:

\[ [x] MINUS \ (POWER \ x \ 2) \ 1 \]

\[ \rightarrow B^2_1 \ MINUS \ ([x] \ POWER \ x \ 2) \ ([x] \ 1) \]

\[ \rightarrow B^2_1 \ MINUS \ (B^2_1 \ POWER \ ([x] \ x) \ ([x] \ 2)) \ ([x] \ 1) \]

\[ \rightarrow B^2_1 \ MINUS \ (B^2_1 \ POWER \ I \ ([x] \ 2)) \ ([x] \ 1) \]

\[ \rightarrow B^2_1 \ MINUS \ (B^2_1 \ POWER \ I \ (K_1 \ 2)) \ ([x] \ 1) \]

\[ \rightarrow B^2_1 \ MINUS \ (B^2_1 \ POWER \ I \ (K_1 \ 2)) \ (K_1 \ 1) \]

Note that the combinatory code is generated in 5 steps whereas Turner's method required 12 steps. The execution of function \( f \) for an argument of 3 results in the following reduction steps:

\[ B^2_1 \ MINUS \ (B^2_1 \ POWER \ I \ (K_1 \ 2)) \ (K_1 \ 1) \ 3 \]

\[ \equiv MINUS \ (B^2_1 \ POWER \ I \ (K_1 \ 2) \ 3) \ (K_1 \ 1 \ 3) \]

\[ \equiv MINUS \ (POWER \ I \ 3 \ (K_1 \ 2 \ 3)) \ (K_1 \ 1 \ 3) \]

\[ \equiv MINUS \ (POWER \ 3 \ (K_1 \ 2 \ 3)) \ (K_1 \ 1 \ 3) \]

\[ \equiv MINUS \ (POWER \ 3 \ 2) \ (K_1 \ 1 \ 3) \]

\[ \equiv MINUS \ 9 \ (K_1 \ 1 \ 3) \]

\[ \equiv MINUS \ 9 \ 1 \]

\[ \equiv 8 \]

As the example we described shows, Abdali's combinatory string requires more steps to reduce to a normal form than does the corresponding Turner's combinatory string. This trend has been found in a number of other examples, too. The reason for this behavior is that Turner uses a much larger set of combinators than Abdali does. However, an important feature of Abdali's combinators is that after every
$B^n_m$ combinator reduction, $n$ independent sub-expressions are generated that are complete (i.e., they have all the data diffused in them) and can be reduced concurrently. This is in contrast to Turner's method where independent sub-expressions are generated after some combinator reductions, but these sub-expressions are not always complete because appropriate data elements may not be present in the sub-expression under consideration.

2.2.3 Hughes' abstraction method

Hughes [11] describes a language translation approach to overcome the disadvantages of Turner's algorithm. Hughes' approach is conceptually different from Turner's and Abdali's in that combinator rules are developed at compile-time. He proposes a compile-time technique for dealing with the problem of free variables in a $\lambda$-abstraction, using a technique known as $\lambda$-lifting. Here, a functional program is translated into a $\lambda$-notation and, starting with the innermost $\lambda$-abstraction, free (full or partial) expressions are lifted from the body and, along with the bound variable of the $\lambda$-abstraction, are made parameters of a combinator, called a super-combinator. This combinator is then substituted in the body of the remaining $\lambda$-expression and the next innermost $\lambda$ is lifted. This process is repeated until the outermost $\lambda$ is lifted. The super-combinator that is generated as a result of lifting the last $\lambda$ represents the program. To reduce a super-combinator code, a modified form of $\beta$-reduction is used in which several $\beta$-reductions may be performed at once.

A super-combinator, $SS$, of arity $n$ is a $\lambda$-expression of the form

$$\lambda x_1 \cdot \lambda x_2 \cdot \ldots \cdot \lambda x_n \cdot E$$

where $E$ is not a $\lambda$-abstraction such that
1. $S$ has no free variables.

2. Any $\lambda$-abstraction in $E$ is a supercombinator.

3. $S$ can be a constant applicative form (CAF), i.e., $n$ can be zero.

For example, $3, + 3 4, \lambda x . x, \lambda x . * * x 2, \lambda x . \lambda y . * (** x 2) y$, and
$\lambda f . f (\lambda x . + x x)$ are all super-combinators, while the following are not:

$$\begin{align*}
\lambda x \cdot f & \quad (f \text{ occurs free}) \\
\lambda x \cdot + x f & \quad (f \text{ occurs free}) \\
\lambda x \cdot x (\lambda f \cdot * f x)
\end{align*}$$

The last $\lambda$-expression is not a super-combinator because $\lambda f$ abstraction is not a super-combinator as $x$ occurs free in it. A formal algorithm for super-combinator generation is as follows:

**WHILE** there is a $\lambda$-abstraction:

(a) Choose any $\lambda$-abstraction which has no inner $\lambda$-abstractions in its body.
(b) Take out all of its free variables or partial/full expressions as extra parameters.
(c) Give an arbitrary name to the $\lambda$-abstraction.
(d) Replace the occurrence of the $\lambda$-abstraction by the name applied to the free variables.
(e) Compile the $\lambda$-abstraction and associate the name with the compiled code.

**END**

During the reduction process, a super-combinator redex, consisting of the application of a super-combinator to $n$ arguments where $n$ is its arity, is chosen and
reduced. A super-combinator reduction replaces a super-combinator redex by an instance of the super-combinator body with arguments substituted for free occurrences of the corresponding formal parameters.

As an example, we take a SASL program, translate it into super-combinators, and reduce it. The example chosen is different from the one used to illustrate Turner's and Abdali's algorithms as a one-variable expression will not illustrate the algorithm. Consider the following function definition:

\[ \text{DEF} \]
\[ f \ x \ y = x \ ** \ 2 + y \ ** \ 2 \]
\[ ? \]

The translation of this function definition into a super-combinator code consists of the following steps. First, the function \( f \) is translated into equivalent \( \lambda \)-notation:

\[ f = \lambda x . \lambda y . + (\ ** x 2) (\ ** y 2) \]

Second, \( \lambda y \) is lifted, resulting in the following super-combinator:

\[ \$R_1 \ P \ y = + P (\ ** y 2) \]

Here, \( \$R_1 \) is an arbitrary name and \( P \) is equal to \( (\ ** x 2) \), which is essentially a free expression when \( \lambda y \) abstraction is considered. The function \( f \) can now be written as:

\[ f = \lambda x \cdot \$R_1 \ P \]
\[ = \lambda x \cdot \$R_1 (\ ** x 2) \]

Now, \( \lambda x \) is lifted to obtain the \( \$R_2 \) super-combinator as given below:

\[ \$R_2 \ x = \$R_1 (\ ** x 2) \]
So, if we make $f$ a super-combinator (denoted by $\$f$) then $\$f = \$R_2$. Thus, if the function $f$ is evaluated with 3 and 4 as its arguments, we have another super-combinator, $\$prog$ say, with the following reduction rule:

$$\$prog = \$f 3 4$$

The reduction of $\$prog$ results in the following steps:

$$\$prog \equiv \$f 3 4$$
$$\equiv \$R_2 3 4$$
$$\equiv \$R_1 (3 \ast \ast 2) 4$$
$$\equiv \$R_1 9 4$$
$$\equiv + 9 (4 \ast \ast 2)$$
$$\equiv + 9 16$$
$$\equiv 25$$

Some of the major advantages of Hughes' method of combinatory code generation are:

1. The translation is faster than Turner's method because all instances of a variable are abstracted in one step. For a program size of $N$, the average and worst-case complexities are $O(N)$ and $O(N \log N)$, respectively.

2. The 'grain' of super-combinators is large, therefore, the percentage overhead of linking one reduction step to the next is smaller as compared to Turner's method.

3. Less transient storage is used.
4. Performance of the reduction machine can be improved by caching the super-combinator bodies.

Some of the disadvantages are:

1. Since the super-combinators are generated at compile time, they cannot be implemented in hardware.

2. The complexity of the underlying reduction machine is more than Turner's reduction machine.

2.3 Detailed Problem Description

Having described three representative methods for compiling a functional program into a combinatory code, we now discuss the functional bottleneck in detail and show how it manifests itself even if we use Abdali's and Hughes' abstraction methods. We accomplish this by taking an example program, translating it into combinatory codes using Turner's, Abdali's, and Hughes' abstraction schemes, and then reducing these codes. For brevity, the complete abstraction process will not be shown here. The set of Abdali's combinators used here is an enhancement over the original set that comprises of $B_n^m$, $K_n$, and $I_n^m$ combinators. The enhanced set has combinators to handle conditionals and incremental list operations and includes: COND, EQ, hd, tl, and CONS. The additional combinators were deemed necessary to accommodate list manipulation and pattern matching.

We choose the matrix multiply program to make our point clear. The reason for this choice is that the matrix multiply algorithm is representative of a wide range of numerical algorithms and allows us to demonstrate the semantics of a
### SASL Script for Matrix Multiply

**DEF**
- `pair a b = a, b`
- `rpair a b = b, a`
- `distl (a, x) = map (pair a) x`
- `distr (a, x) = map (rpair x) a`
- `map f x = x() -> (); f (hd x) : map f (tl x)`
- `zip x = (hd x)() -> (); map hd x : zip (map (tl x))`
- `IP x = sum (map product (zip x))`
- `mult x y = map (map IP) (map distl (distr (x, zip y)))`
- `program = mult ((1,2),(3,4))((1,2),(3,4))`

### Turner's Combinatory Code

- `pair = C B_p (C_p I NIL)`
- `rpair = B (C_p I) (C_p I NIL)`
- `distl = U_1 (B U_1 (B (MATCH NIL)) map pair))`
- `distr = B_1 U_1 (B (MATCH NIL)) (C_1 map rpair))`
- `map = B (S (C_1 COND (C EQ NIL)) (C_1 map hd) (C_1 B map tl))`
- `zip = S (C_1 COND (C_1 EQ hd NIL)) (S_p (map hd) (B_1 zip (map tl)))`
- `IP = B_1 sum (map product) zip`
- `mult = B_1 (B (map (map IP)) (B_1 (map distl) distr) (C B_p (C_p zip NIL))`
- `program = mult ((1,2),(3,4))((1,2),(3,4))`

### Abdali's Combinatory Code

- `pair = CONS`
- `rpair = B_3 (CONS I_3 I_3)`
- `distl = B_3 map (B_3 pair hd) tl`
- `distr = B_3 map (B_3 rpair tl) hd`
- `IP = B_3 sum (B_3 map product zip)`
- `map = B_3 COND (B_3 I_3 (X_3 NIL))(X_3 NIL)(B_3 COND (B_3 I_3 (K_1 hd))(B_3 map I_3 (X_1 tl)))`
- `zip = B_3 COND (B_3 I_3 (X_1 hd))(B_3 map I_3 (X_1 tl))(B_3 COND (B_3 I_3 (K_1 NIL))(X_1 NIL)(B_3 COND (B_3 I_3 (map hd)(B_3 zip (map tl)))`
- `mult = B_3 map (map IP)(B_3 map distl (B_3 distr (B_3 I_3 (K_1 zip)))`
- `program = mult ((1,2),(3,4))((1,2),(3,4))`

### Super-Combinator Code

- `$pair a = SB A$
- `$SB A b = A b$
- `$rpair a b = CONS b a$
- `$Sdistl x = $map ($pair (hd x))(tl x) `$
- `$Sdistr x = $map ($rpair (tl x))(hd x)`
- `$Smap f x = COND (EQ x NIL) NIL (CONS (f (hd x)) ($map f (tl x)))`
- `$Szip x = COND (EQ (hd x) NIL) NIL (CONS ($map (hd x)) ($zip ($map (tl x)))`
- `$SIP x = $sum ($map $product ($zip x))$
- `$Smult x y = $map ($map SIP) ($map $distr ($distr (x ($zip y))))$
- `$$prog = $mult ((1,2),(3,4))((1,2),(3,4)`)`

Figure 2.1: SASL script and combinatory codes for matrix multiply
variety of list operations. In the later part of this dissertation, these and other representative list operations will be used to define a set of list combinators to efficiently execute functional languages that are implemented using a combinator-based graph reduction model. The SASL script and different combinatory codes for matrix multiply are given in Figure 2.1.

The reduction of these combinatory codes reveals that if parallelism is available at the program level, it can be exploited quickly if Abdali's or Hughes' algorithm is used. Abdali's combinators give us an additional advantage in that they also act as control mechanism indicating the position of those sub-expressions that can be reduced concurrently. In the case of Hughes' combinators, on the contrary, a separate mechanism is needed to flag those sub-expressions that can be reduced in parallel after a reduction step. However, whatever abstraction and reduction mechanism is employed, list manipulation remains recursive and sequential. The reason for this recursive list manipulation is that none of the abstraction methods performs abstraction on elements of the list. This point will be made clear by reducing the combinatory code for a standard list processing function. Although the intention was to show complete reduction steps for the reduction of all three combinatory strings (Turner's, Hughes', and Abdali's) for the matrix multiply example, it was not feasible due to the large number of reduction steps involved in their reduction. For example, the reduction of Turner's code needs 1061 reductions to multiply two 2x2 matrices. However, it seems sufficient to show reduction steps for a list processing function used in the matrix multiply example. The map function is chosen to make our point.

The reduction steps for the map function when used to apply a function (plus
Figure 2.2: First 33 reduction steps for \(map\) \((plus\ 1)\) \((1,2)\) using Turner's combinatory code
Figure 2.3: First 13 reduction steps for map (plus 1) (1,2) using Abdali's combinatory code
1) to (1,2) are shown in Figures 2.2 and 2.3 for Turner’s and Abdali’s combinatory strings, respectively. Notice that the \textit{map} function applies (plus 1) to the first element of the list, returns the result of this application (which is 2 in our example), and then applies (plus 1) to the remaining list, (2,). This is clearly shown by the first 20 reduction steps in the case of Turner’s string and first 13 steps in the case of Abdali’s string. The combinatory string comprising of Hughes’ super-combinators, as given in Table 2.1, can clearly be seen as having the same semantics: manipulating the list recursively by applying a given function to the first element of the list and then calling \textit{map} to apply the function to the remaining list. The point is that even though Abdali’s and Hughes’ abstraction methods diffuse the actual parameters into the combinatory code in such a manner that there are opportunities for parallel reduction of sub-expressions, lists are substituted into the code as any other variable. Consequently, the \textit{map} function is executed for every element of the list in a sequential fashion. Therefore, although every instance of the \textit{map} function is applied to a distinct list element and the parallelism, which intuitively seems to be there, cannot be exploited because:

1. Abstraction methods do not perform abstraction on list elements.

2. The semantics of list representation are such that the indexing (random access) of list elements is not possible.

Thus, any function that is written to randomly access list elements recursively scans the list until the required element is reached. What this means is that to have a true concurrent access capability in a functional language, the underlying implementation should incorporate the following:
1. A set of combinators whose semantics allow for random and concurrent access of list elements.

2. A list representation that accommodates the above combinators.

In the next three chapters, a set of combinators and a list structure that allow random and simultaneous access of list elements are described. The next chapter introduces the notation and terminology that is used in the remaining chapters.
3 TERMINOLOGY, NOTATION, AND PRELIMINARY COMBINATORS

3.1 Introduction

In this chapter, we introduce notation, terminology, and some preliminary combinators that are used as basic building blocks for the list combinators described in the next chapter. Some of the terminology and combinators described here already exist in the literature [6]. Additional notation and combinators are introduced to help develop the major combinators which are the main topic of this dissertation. A few operations are defined on the preliminary combinators in order to achieve a compact notation for the new families of combinators. The significance and intuitive meanings of the notation and operations on combinators are, wherever possible, given as they are introduced.

3.2 Terminology and Notations

In this section, definitions and terminology used in the remaining chapters are discussed. Most of the terminology and notation described in this section is taken from [6]. However, additional terminology is introduced in order to achieve concise notation for the families of list combinators described in the next chapter.
Definition 1: A proper combinator, also known as a pure combinator, is one which corresponds to a combination of variables only.

S, C, B, K, and I combinators are examples of proper combinators. An example of an improper combinator could be \( C I B \) because

\[
C I B f \ x_1 \ x_2 \ldots \ x_n \equiv f \ B \ x_1 \ x_2 \ldots \ x_n
\]

Note that the right-hand-side of the above equation involves the B combinator and is, therefore, not a combination of variables \((f, x_1, x_2, \text{etc.})\) only. The importance of proper combinators is that the combinatory code comprising of such combinators can be reduced using very simple rules, for example, normal-order reduction scheme.

Definition 2: A regular combinator is a proper combinator in whose reduction rule the first variable remains in the first position without modification.

S, C, B, K, and I are examples of regular combinators. The set of list-manipulation combinators described in the following chapters are regular. To make these combinators regular was a conscious decision on the part of the author. This decision was made after observing that non-regular combinators cannot be used to generate a combinatory code that can be reduced in normal-order without introducing extra combinators and incorporating a complicated code generation mechanism, thereby making the code unnecessarily long.

Definition 3: The order of a combinator is the number of arguments it needs to perform a reduction step.
For example, the order of I is 1, the order of K is 2, and the order of B, C, and S is 3. The order of a combinator comes into consideration while doing its (time and space) complexity analysis.

**Definition 4:** For any combinatory terms X and Y,

\[ X.Y \equiv B \times Y \]

The dot (.) operation is called the *composite product* and is associated to the left, so that

\[ X.Y.Z \equiv (X.Y).Z \]

Therefore, \( C.C \equiv B.C.C \), \( C.(B.C) \equiv B.C(B.C) \), etc.

**Definition 5:** For any combinatory term X and natural number n, the *powers* operation is defined as:

\[
\begin{align*}
X^0 & \equiv I \\
X^1 & \equiv X \\
X^{n+1} & \equiv X \times X^n
\end{align*}
\]

Therefore, \( C^2 \equiv C.C \), \( B^3 \equiv B.B.B \), etc.

**Definition 6:** For any combinatory term X and any natural number n we define \( X(n) \) recursively:

\[
\begin{align*}
X(0) & \equiv X \\
X(n+1) & \equiv B \times X(n)
\end{align*}
\]
Here, juxtaposition stands for application and it is right-associative. Thus, $X(2) \equiv B(B\ X)$, $X(3) \equiv B(B(B\ X))$, etc. Intuitively, $X(n)$ defers the effect of $X$ by $n$ steps. Thus, $C(n)$ interchanges $n+2$nd and $n+3$rd arguments; $W(n)$ causes repetition of $n+2$nd argument; and $K(n)$ causes cancellation of the $n+2$nd argument.

**Definition 7:** For any combinatory term $X$, the star (*) operation is defined as: 

$$X_* \equiv X \ I$$

Thus, $C_* \equiv C \ I$, $(BC)_* \equiv (B \ C) \ I$, and $K^n_* \equiv K^n \ I$, etc.

**Definition 8:** For any combinatory terms $X$ and $Y$, $X[n]$ is recursively defined as:

$$X[1] \equiv X$$
$$X[n] \equiv X^{n-1}X$$
$$X[n]Y \equiv X^{n-1}(XY)$$

Here again, juxtaposition stands for application, and application is right-associative.

Therefore, $B[2] \equiv B[1] \ B \equiv B \ B$, $B[3] \equiv B[2] \ B \equiv B \ (B \ B)$, $B[3] \ C \equiv B \ (B \ (B \ C))$, etc.

**Definition 9:** For any combinatory term $X$, we have the following recursive definition:

$$(X)_0^m \equiv I$$
$$(X)_1^m \equiv BX_m$$
$$(X)_n^m \equiv (X)^{n-1}_m (B^n X_m)$$
Thus, for example,

\[(C)_{\frac{3}{2}} \equiv (C)_{\frac{2}{2}} (B^{[3]} C_2) \equiv BC_2 (B(B C_2) (B(B(B C_2))))\]

Therefore, for any combinatory terms \(X\) and \(Y\),

\[(X)_{\frac{n}{m}} Y \equiv B^{[1]} X_m (B^{[2]} X_m (B^{[3]} X_m \ldots (B^{[n]} X_m Y) \ldots ))\]

**Definition 10:** For any combinatory sequence \(X_1 \ldots X_n\) and a combinatory term \(Y\),

\[(X_1 \ldots X_n)_{\frac{m}{m}} Y \equiv (X_1)_{\frac{m}{m}} ((X_2)_{\frac{m}{m}} \ldots ((X_n)_{\frac{m}{m}} Y) \ldots )\]

Thus, for example, \((H O Z)_3 B \equiv H_3 (O_3 (Z_3 B)).\)

**Definition 11:** Using definitions 9 and 10, we obtain the following inductive definition:

\[(X_1 \ldots X_k)_{\frac{n}{m}} \equiv (X_1 \ldots X_k)_{\frac{n-1}{m}} (B^{[n]} (X_1 \ldots X_k)_{\frac{m}{m}} ) \equiv B^{[1]} (X_1 \ldots X_k)_{\frac{m}{m}} (B^{[2]} (X_1 \ldots X_k)_{\frac{m}{m}} \ldots (B^{[n]} (X_1 \ldots X_k)_{\frac{m}{m}} ) \ldots )\]

For example,

\[(H O Z)_3 \equiv B^{[1]} (H O Z)_{\frac{3}{3}} (B^{[2]} (H O Z)_3) \equiv B^{[1]} H_3 (B^{[1]} O_3 (B^{[2]} (B^{[2]} (B^{[2]} (B^{[2]} (B^{[2]} Z))) ) ) ) ) ) )

Definitions 4 through 8 give very powerful notations which will be apparent when equivalent combinatory strings for the list-combinators are given in the next chapter.
3.3 Preliminary Combinators

This section introduces some basic combinators from the standpoint of their reduction rules. We start with the reduction rules for some well-known combinators that have been discussed in [6,1,4]. In the remainder of this dissertation, these combinators are referred to as the basic combinators. Following are the reduction rules for these combinators:

\[
\begin{align*}
I x & \rightarrow x \\
K x y & \rightarrow x \\
W x y & \rightarrow x y y \\
D x y z & \rightarrow z \\
B x y z & \rightarrow x (y z) \\
C x y z & \rightarrow (x z) y \\
S x y z & \rightarrow (x z) (y z) \\
K_n f x_1 \ldots x_n & \rightarrow f
\end{align*}
\]

All of these combinators, except D and \(K_n\), are well-known. The D combinator essentially takes three arguments and returns the third argument. The \(K_n\) combinator has a degree of \(n+1\) and it returns the first argument. The \(K\) combinator is essentially a limited case of \(K_n\) where \(n=1\).

Now, we define a set of new regular combinators. These combinators will help develop the equivalent combinatory strings for the parallel list-combinator families discussed later in the dissertation. Following are the reduction rules for the new
families of combinators:

\[
\begin{align*}
L_n f g x_1 x_2 \ldots x_n y & \to f g (y x_1)(y x_2)\ldots(y x_n) \\
M_n f g x y_1 y_2 \ldots y_n & \to f g x y_1 x y_2 x y_3 x \ldots x y_n \\
N_n f g y_1 y_2 \ldots y_n x & \to f g y_1 x y_2 x x y_3 x \ldots x y_n x \\
O_m f x_1 x_2 x_3 \ldots x_n & \to f x_1 x_3 x_5 x_m x_2 x_4 x_n
\end{align*}
\]

The \( L_n \) combinator family has a degree of \( n+3 \). Intuitively, the reduction of this combinator results in \( n \) two-element combinations. Each of these combinations is composed of the last \((n+3)\)rd argument and an argument starting from the third argument. For example,

\[
L_2 f g x y z = (z x)(z y).
\]

The second combinator, \( M_n \), has a degree of \( n+3 \) and its reduction results in a combination of its arguments with the third argument occupying odd-numbered positions starting with the third position. For example, \( M_3 f g x a b c = f g x a x b x c \) and \( M_2 f g * (1,2) (3,4) = f g * (1,2) * (3,4) \).

The \( N_n \) combinator family also has a degree of \( n+3 \). The reduction semantics of this family are similar to the \( M_n \) family: reduction of a combinator in this family results in a combination of its arguments that has the last argument distributed at even-numbered positions, starting with the fourth position. For example, \( N_2 f g 1 2 3 = f g 1 3 2 3 \), \( N_2 f g (1,2) (3,4) * = (1,2) * (3,4) * \), etc.

Finally, the \( O_m \) combinator family has a degree of \( n+1 \) and it makes a combination of all odd-numbered arguments followed by all even-numbered arguments.

\[
\begin{align*}
O_2 f x_1 x_2 x_3 x_4 & \equiv f x_1 x_3 x_2 x_4 \\
O_3 f x_1 x_2 x_3 x_4 x_5 & \equiv f x_1 x_3 x_5 x_2 x_4
\end{align*}
\]

Here, \( n \leq 2m \) if \( n \) is even and \( n \leq 2m-1 \) if \( n \) is odd. For example, \( O_2 f x_1 x_2 x_3 x_4 \equiv f x_1 x_3 x_2 x_4 \) and \( O_3 f x_1 x_2 x_3 x_4 x_5 \equiv f x_1 x_3 x_5 x_2 x_4 \).
By using the definitions of powers and deferred combinators, we define a family of combinators, \((K_{(m)})^n\), that deletes \(n\) consecutive arguments starting from the \(m+2\)nd argument. This combinator family has a degree of \((m+n+1)\) and the following reduction rule:

\[
(K_{(m)})^n f \, x_1 \ldots x_m \ldots x_{m+n} \rightarrow f \, x_1 \ldots x_m
\]

Following the definitions of powers, composite product, and delayed combinators, we have, for example

\[
(K_{(2)})^2 \equiv K_{(2)} \cdot K_{(2)}
\]

\[
\equiv B K_{(2)} K_{(2)}
\]

\[
\equiv B(B(B \, K))(B(B \, K))
\]

To illustrate the effect of this combinator family, complete reduction process for one member of the family, namely \((K_{(2)})^2\), is given.

\[
(K_{(2)})^2 x_1 \ldots x_6
\]

\[
\equiv B(B(B \, K))(B(B \, K)) \, x_1 \ldots x_6
\]

\[
\equiv B(B \, K)(B(B \, K) \, x_1) \, x_2 \ldots x_6
\]

\[
\equiv B \, K(B(B \, K) \, x_1 \, x_2) \, x_3 \ldots x_6
\]

\[
\equiv K(B(B \, K) \, x_1 \, x_2 \, x_3) \, x_4 \ldots x_6
\]

\[
\equiv B(B \, K)x_1 \, x_2 \, x_3 \, x_4 \, x_5 \, x_6
\]

\[
\equiv B \, K(x_1 \, x_2) \, x_3 \, x_5 \, x_6
\]

\[
\equiv K(x_1 \, x_2 \, x_3) \, x_5 \, x_6
\]

\[
\equiv x_1 \, x_2 \, x_3 \, x_6
\]

As it has been mentioned earlier, the \(C_{(n)}\) combinator family exchanges the positions of the \(n+2\)nd and the \(n+3\)rd arguments. The reduction rule for this
combinator family is:

\[ C(n) f x_1 \cdots x_n x_{n+1} x_{n+2} \cdots x_k \rightarrow f x_1 \cdots x_n x_{n+2} x_{n+1} \cdots x_k \]

For example, \( C(3) x_1 x_2 x_3 x_4 x_5 x_6 \cdots x_m \equiv x_1 x_2 x_3 x_4 x_5 \cdots x_m \). Now, the complete reduction process for one member of this family, \( C(3) \), is shown to illustrate the semantics of its reduction. The definition of delayed combinators is used to expand \( C(n) \):

\[
\begin{align*}
C(3) x_1 \cdots x_m & \\
& \equiv B(B(B C)) x_1 \cdots x_m \\
& \equiv B(B C)(x_1 x_2) x_3 \cdots x_m \\
& \equiv B C(x_1 x_2 x_3) x_4 \cdots x_m \\
& \equiv C(x_1 \cdots x_4) x_5 \cdots x_m \\
& \equiv x_1 \cdots x_4 x_6 x_5 x_7 \cdots x_m
\end{align*}
\]

Finally, a new notation, \( C_{(m,n)} \), is introduced. When applied to any combinatory term \( X \),

\[ C_{(m,n)} X \equiv C_{(m)}(C_{(m+1)}(C_{(m+2)} \cdots (C_{(m+n-1)} X) \cdots)) \]

For example, \( C_{(3,3)} B \equiv C_{(3)}(C_{(4)}(C_{(5)} B)) \).

In the next chapter, the new families of list-combinators are proposed. Throughout the discussion, the above-described notations and combinators and their extensions are used to show the reduction semantics and proofs of correctness of the proposed combinators.
4 COMBINATORS FOR LIST MANIPULATION

4.1 Introduction

In this chapter, new list-combinators are introduced. These combinators express certain commonly occurring combinations of lists or elements of lists as functions of those lists or elements of lists. A reduction rule associated with each of the combinator families states that when a combinator is applied to a list with a (finite) number of elements, the resulting combination reduces to a certain combination of the list elements. An efficient implementation of these combinators demands a list representation that is different from the conventional recursively defined list structure. In this dissertation, however, the reduction semantics of the new combinators are represented as reductions of equivalent strings of already known combinators.

The only true data structure in all functional languages is a list. Conventionally, physical realization of the list structure consists of a linked-list of list cells. A list cell is composed of a pair of pointers: a pointer to the contents of this cell and a pointer which is a link to the next list cell. All functional programming languages have a set of predefined primitive functions to manipulate lists incrementally. This set includes $hd$ or $car$ (returns the first element of a list), $tl$ or $cdr$ (returns everything following the first element of a list), and $cons$ (creates a new list cell and appends it to the head of the given list). During the execution of a functional language
program, manipulation of lists often involves traversing several pointers at run-time using this set of primitive functions. A large fraction of execution time is spent in following these pointers [7,15]. For example, there is only one basic list operator which appends an element at the head of a list. Thus, concatenation of two lists is carried out by scanning the first list until the end is encountered and then, starting from the last element, each element is appended to the second list one at a time. It would be much more efficient to have an operator that directly (in one step) puts an element at the end of a list. Such an operator is avoided because it destroys the property of sharing a list, which essentially is a direct consequence of the recursive representation of list structure, as discussed above.

A consequence of the recursive list representation and lack of list operations is that parallel algorithms that involve vector, list, and array processing cannot be efficiently executed in a multiprocessing processing environment. A study by Hahn [7] shows that for “real world” functional programs (e.g., database management programs), almost 80% of the program execution time is spent on reshaping the list structure. Considering such a high frequency of list operations during the execution of a functional program, it seems imperative to have a set of list-manipulation combinators, a new representation of the list structure, and new list operators to perform parallel operations on a list.

In the following sections, a set of combinator families is introduced to efficiently perform some basic list operations. These families of combinators are then used to define another set of combinator families that efficiently performs higher-level list operations like transpose, reversal of a list, etc. The purpose of these combinators is to reshape a given list for concurrent access of its elements. Finally, a set of list
combinators is introduced to perform \textit{map}-like parallel operations on elements of a list. The equivalence of these combinators in terms of strings of already known combinators is also given. The reduction semantics for these equivalent combinator strings, though sequential, represent the proofs of correctness of the proposed combinator families.

4.2 Basic List Combinators

In this section, a set of list combinators is defined that is the basis for the list-manipulation combinators discussed in the next section. The combinators are essentially list constructors and destructors and have the following reduction rules:

\begin{align*}
P_n f g x_1 \ldots x_{n-1} x_n & \to f g x_1 \ldots x_{n-1} (x_n) \\
U^{(n)} f g (x_1, \ldots, x_p) & \to f g x_1 \ldots x_n (x_{n+1}, x_{n+2}, \ldots, x_p) \quad p \geq n \\
J^{(n)} f g x_1 \ldots x_{n-1} (l_1, \ldots, l_m) & \to f g (x_1, \ldots, x_{n-1}, l_1, \ldots, l_m) \quad m \geq 0 \\
Z_{k,m} f g x_1 \ldots x_m x_{m+1} \ldots x_{m+k-1} & \to f g x_1 \ldots x_{m-1} (x_m, \ldots, x_{m+k-1}) \\
H_m f L_1 L_2 \ldots L_m & \to f (\mathrm{hd} \ L_1) (\mathrm{tl} \ L_1) (\mathrm{hd} \ L_2) (\mathrm{tl} \ L_2) \ldots (\mathrm{hd} \ L_m) (\mathrm{tl} \ L_m) \\
Z_m f x_1 \ldots x_m & \to f (x_1 \ldots x_m)
\end{align*}

The first of these combinators, $P_n$, has a degree of $n+2$ and it makes a singular list of the last argument. Hence, $P_2 f g 1 2 \equiv f g 1 (2,)$, $P_4 C B 1 2 3 4 \equiv C B 1 2 3 (4,)$, etc.

The second combinator family is $U^{(n)}$ with a degree of 3. Intuitively, $U^{(n)}$ takes a list of $p$ elements as an argument, pulls out the first $n$ elements ($p \geq n$), and returns these $n$ elements along with the rest of the list. For example,
\( U^{(3)} f g (1,2,3) \equiv f g 1 2 3 () \) and \( U^{(3)} f g (1,2,3,4) \equiv f g 1 2 3 (4,) \).
For any combinatory term \( X \), \( U^{(n)} \) can be defined inductively:

\[
U^{(1)} X \equiv B[1] U X \\
U^{(n)} X \equiv U^{(n-1)} (B[n] U X)
\]

where \( U \) is a combinator that is used by Turner in his SASL compiler. We can see that \( U \) is a list selector combinator that selects the first element of the input list and is defined by the following reduction rule:

\[
U f (a : b) \rightarrow f a b
\]

Hence,

\[
\]

Thus,

\[
U^{(2)} f g (1,2) \\
\equiv U^{(1)} (B[2] U f) g (1,2) \\
\equiv B U (B (B U) f) g (1,2) \\
\equiv U (B (B U) f g) (1,2) \\
\equiv B (B U) f g 1 (2,) \\
\equiv B U (f g 1) (2,) \\
\equiv U (f g 1) (2,) \\
\equiv f g 1 2 ()
\]

The third of the basic list combinators is \( J^{(n)} \). It has a degree of \( n+2 \) and appends the third through \( n+1 \) arguments to the \( n+2 \)nd argument in order, where the \( n+2 \)nd argument is a list. For example, \( J^{(3)} I I 1 2 (3,) \equiv I I (1,2,3) \) and
\( J^{(3)} f \ g \ 1 \ 2 \ ) \equiv f \ g \ (1, 2) \). For any combinatory term \( X \),

\[
J^{(n)} X \equiv J_n (J_{n-1} (J_{n-2} \cdots (J_1 (J_0 X)) \cdots))
\]

and \( J^{(1)} \equiv J^{(0)} \equiv I \). The \( J_k \) combinator appends the \( k+1 \)st element to the \( k+2 \)nd element (which is a list). The reduction rule for \( J_k, k \geq 1 \), is therefore

\[
J_k f \ x_1 \ldots x_{k-1} \ x_k (l_1, \ldots, l_m) \rightarrow f \ x_1 \ldots x_{k-1} (x_k, l_1, \ldots, l_m)
\]

\( J_1 \equiv J_0 \equiv I \). We show the complete reduction process for \( J^{(3)} \) to verify its correctness. The definition of \( J_k \) is used to explain the reduction semantics.

\[
J^{(3)} I \ I \ I \ 1 \ 2 \ (3,)
\]

\[
\equiv J_3 (J_2 (J_1 (J_0 I))) \ I \ I \ 2 \ (3,)
\]

\[
\equiv J_2 (I (I I)) \ I \ I \ (2, 3)
\]

\[
\equiv I (I I) \ I \ (1, 2, 3)
\]

\[
\equiv I \ I \ I \ (1, 2, 3)
\]

\[
\equiv I \ I \ I \ (1, 2, 3)
\]

\[
\equiv I \ (1, 2, 3)
\]

\[
\equiv (1, 2, 3)
\]

The fourth combinator family, \( Z_{k,m} \), has a degree of \( k+m+1 \). This family makes a list of \( k \) of its arguments starting from the \( m+2 \)nd argument. Thus, \( Z_{3,5} f \ g \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \equiv f \ g \ 1 \ 2 \ 3 \ 4 \ (5, 6, 7) \ 8 \). In terms of the already discussed combinators and notation,

\[
Z_{k,m} \equiv B \ P_{m+k-1} (B^{[m-1]} J(k))
\]
To demonstrate the correctness of this equivalence, complete reduction sequence for \( Z_{3,5} \) is given:

\[
Z_{3,5} f g 1 2 3 4 5 6 7 8
\]

\[
\rightarrow BP_{5+3-1} (B^{[5-1]} J^{(3)}) f g 1 2 3 4 5 6 7 8
\]

\[
\rightarrow P_7 (B^{[4]} J^{(3)}) f g 1 2 3 4 5 6 7 8
\]

\[
\rightarrow B^{[4]} J^{(3)} f g 1 2 3 4 5 6 (7,) 8
\]

\[
\rightarrow J^{(3)} (f g 1 2 3) 4 5 6 (7,) 8
\]

\[
\rightarrow f g 1 2 3 4 (5,6,7) 8
\]

The fifth family, \( H_m \), takes m lists as arguments and returns a combination of heads and tails of all the lists. Here, each \( L_i \) (1 \( \leq \) i \( \leq \) m) is a non-empty list. For example, \( H_2 f (1,2,3) (4,5,6) \equiv f 1 (2,3) 4 (5,6) \) and \( H_3 f (1,2) (3,4) (5,6) \equiv f 1 (2) 3 (4) 5 (6) \). If each list is represented as (hd:tl),

\[
H_0 \equiv I
\]

\[
H_1 \equiv BU I
\]

\[
H_m \equiv B^{[1]}U(B^{[3]}U(B^{[5]}U(B^{[7]}U\ldots(B^{[2m-1]}U I)\ldots)))
\]

Thus,

\[
H_2 f (a : b) (c : d)
\]

\[
\equiv B^{[1]}U(B^{[3]}U I) f (a : b) (c : d)
\]

\[
\equiv BU(B(B(B U)) I) f (a : b) (c : d)
\]

\[
\equiv U(B(B(B U)) I f) (a : b) (c : d)
\]

\[
\equiv B(B(B U)) I f a b (c : d)
\]

\[
\equiv B(B U) (I f) a b (c : d)
\]

\[
\equiv BU (I f a) b (c : d)
\]

\[
\equiv U (I f a b) (c : d)
\]
The sixth family, \( Z_m \), takes \( m+1 \) arguments and returns a list of the second through \( m+1 \) arguments. For example, \( Z_2 \) \( I \ 1 \ 2 \equiv (1,2) \) and \( Z_3 \) \( I \ 1 \ 2 \ 3 \ 4 \equiv (1,2,3) \ 4 \). Using the basic list combinators, we obtain the following relation:

\[
Z_m \equiv B^{[m]} P_1(J^{(m)}_*)
\]

For example, the following reduction steps are obtained when we apply \( Z_4 \) to \( 1 \ 2 \) \( \ldots \ n \):

\[
Z_4 \ f \ 1 \ 2 \ \ldots \ n
\equiv B^{[4]} P_1(J^{(4)}_*) \ f \ 1 \ 2 \ \ldots \ n
\equiv P_1((J^{(4)}_*) \ f \ 1 \ 2 \ 3) \ 4 \ \ldots \ n
\equiv J^{(4)}_* \ f \ 1 \ 2 \ 3 \ (4,) \ 5 \ \ldots \ n
\equiv J^{(4)} \ I \ f \ 1 \ 2 \ 3 \ (4,) \ 5 \ \ldots \ n
\equiv I \ f \ (1,2,3,4) \ 5 \ \ldots \ n
\equiv f \ (1,2,3,4) \ 5 \ \ldots \ n
\]

In the following section, some list combinators are introduced to perform a representative set of operations on list. These operations include list indexing, transpose of a list of lists, constructing a list of lists out of a list, etc.

### 4.3 Major List-Manipulation Combinators

In this section, another set of combinators is introduced to perform higher-level list manipulation. The set of these operations has been chosen to accommodate
most of the list applications. Some of these operations, like random access of list elements, are not supported in functional languages because of the way list manipulation is carried out in the functional approach. However, the inclusion of such operations in a functional language would tremendously enhance its execution speed on current processor architectures and memory organizations. The equivalence of these combinators is presented as strings of already discussed combinators.

4.3.1 List indexing

A family of combinators, \( D_{(m)} \), is defined to index a list. This combinator returns the \( m^{th} \) element from a list. The reduction rule for this family is given below:

\[
D_{(m)} f (x_1, \ldots, x_n) \rightarrow f x_m \quad n \geq m
\]

For example, \( D_{(3)} f (1, 2, 3) \equiv 3 \) and \( D_{(2)} f (1, 2, 3, 4) \equiv 2 \). In terms of the already discussed combinators and notations, \( D_{(m)} \) can be defined as:

\[
D_{(m)} \equiv U(m) \ (W_{(m)} \ (C_{(m+1)} \ (J^{(m+1)} \ D)))
\]

To see the validity of this equivalence, the complete reduction process for \( D_{(3)} \), which returns the third element of a list of three elements, is presented:

\[
\begin{align*}
D_{(3)} f (1, 2, 3) & \equiv U(3) (W_{(3)} (C_{(3+1)} (J^{(4)} \ D))) f (1, 2, 3) \\
& \equiv W_{(3)} (C_{(4)} (J^{(4)} \ D)) f 1 2 3 () \\
& \equiv C_{(4)} (J^{(4)} \ D) f 1 2 3 3 () \\
& \equiv J^{(4)} \ D f 1 2 3 () 3
\end{align*}
\]
4.3.2 Taking elements from the head of a list

A combinator, \( H^{(m)} \), is defined to return \( m \) elements from the head of a list according to the following reduction rule:

\[
H^{(m)} f (x_1, \ldots, x_{m}, x_{m+1}, \ldots, x_n) \rightarrow f x_1 \ldots x_m \quad m \leq n
\]

Hence, \( H^{(3)} I (1, 2, 3, 4) = I 1 2 3 \equiv 1 2 3 \). Using the already defined combinators and notations, we obtain:

\[
H^{(m)} \equiv U^{(m)} (B^{[m]} K) \quad m \leq |\text{list}|
\]

For example, \( H^{(3)} \equiv U^{(3)} (B^{[3]} K) \equiv U^{(3)} (B (B (B K))) \). The complete reduction of \( H^{(3)} \) that returns three elements from the head of a list is presented to illustrate its correctness:

\[
H^{(3)} f (1, 2, \ldots, n) \\
\equiv U^{(3)} (B^{[3]} K) f (1, 2, \ldots, n) \\
\equiv B^{[3]} K f 1 2 3 (4, \ldots, n) \\
\equiv B (B (B K)) f 1 2 3 (4, \ldots, n) \\
\equiv B (B K) (f 1) 2 3 (4, \ldots, n) \\
\equiv B K (f 1 2) 3 (4, \ldots, n) \\
\equiv K (f 1 2 3) (4, \ldots, n) \\
\equiv f 1 2 3
\]
4.3.3 Taking elements from the tail of a list

A combinator family is defined that returns $m$ elements from the tail of a list of $n$ elements. This combinator family is $T^{(m,n)}$ and is defined by the following reduction rule:

$$ T^{(m,n)} f (x_1, \ldots, x_{n-m}, x_{n-m+1}, \ldots, x_n) \rightarrow f x_{n-m+1} \ldots x_n $$

For example, $T^{(2,5)} f (1, 2, 3, 4, 5) \equiv 45$. $T^{(m,n)}$ can be defined as

$$ T^{(m,n)} \equiv T_n (T^{n-1} \ldots (T_{n-m+1} D) \ldots) $$

where the $T_n$ combinator chops off the last element from a list. $T_n$ is defined by the following reduction rule:

$$ T_n f g (x_1, \ldots, x_n) \equiv f g (x_1, \ldots, x_{n-1}) x_n $$

For example, $T^{(2,5)} \equiv T_5 (T_4 D)$. The reduction process for $T^{(2,5)}$, that returns the last two elements of a five-element list, is given to illustrate the reduction semantics of this family. Assume the input list is $(1,2,3,4,5)$.

$$ T^{(2,5)} f (1, 2, 3, 4, 5) $$

$$ \equiv T_5 (T_4 D) f (1, 2, 3, 4, 5) $$

$$ \equiv T_4 D f (1, 2, 3, 4) 5 $$

$$ \equiv D f (1, 2, 3) 4 5 $$

$$ \equiv 45 $$

4.3.4 List expansion (insertion)

A combinator family is defined that expands a list by inserting one or more elements in it starting from some given element. The family is $(n,k)C$ with the
following reduction rule:

\[(n,k)C \, f \, (x_1, \ldots, x_{n-1}, x_n, \ldots, x_p) \, x_i \ldots x_{i+k-1} \rightarrow f \, (x_1, \ldots, x_{n-1}, x_i, \ldots, x_{i+k-1}, x_n, \ldots, x_p)\]

The \((n,k)C\) combinator has a degree of \(k+2\) and inserts \(k\) elements into the list starting at the \(n^{th}\) element. The second argument is a list and the third to \(k+2\) arguments are the items to be inserted in the list. Here, \(k, n \geq 1\) and \(p \geq n\). For example, \((4,2)C \, I \, (1, 2, 3, 4) \, 5 \, 6 \equiv I \, (1, 2, 3, 5, 6, 4)\). In terms of the existing combinator and notations,

\[(n, k)C \equiv U^{(n-1)} \, (C_{(n,k)} \, J_\times^{(n+k)})\]

Thus \((4,2)C \equiv U^{(3)} \, (C_{(4,2)} \, J_\times^{(6)}) \equiv U^{(3)} \, (C_{(4)} \, (C_{(5)} \, (J^{(6)} \, I)))\). To insert a single element into the list at the \(n^{th}\) position, \(k=1\) in the reduction rule. Therefore,

\[(n, 1)C \equiv U^{(n-1)} \, (C_{(n-1,1)} \, (J_\times^{(n+1)}))\]

Consider inserting 8 and 9 at the fourth and fifth positions in \((1, 2, \ldots, n)\). In this case, \(k=2\) and \(n=4\), therefore, \((4,2)C\) combinator is used. Following is the complete reduction sequence:

\[(4,2)C \, f \, (1, 2, \ldots, n) \, 8 \, 9 \equiv U^{(3)}(C_{(3,2)} \, J_\times^{(6)}) \, f \, (1, 2, \ldots, n) \equiv U^{(3)}(C_{(4)}(C_{(5)}((J^{(6)} \, I)))) \, f \, (1, 2, \ldots, n) \, 8 \, 9 \equiv C_{(4)}(C_{(5)}(J^{(6)} \, I)) \, f \, 1 \, 2 \, 3 \, (4, \ldots, n) \, 8 \, 9 \equiv C_{(5)}(J^{(6)} \, I) \, f \, 1 \, 2 \, 3 \, 8 \, (4, \ldots, n) \, 9\]
\[ \equiv J(6) f 1 2 3 8 9 (4, \ldots, n) \]
\[ \equiv f (1, 2, 3, 8, 9, 4, \ldots, n) \]

4.3.5 Deletion of list elements

In this section, we introduce a combinator family, \((K)[m,n]\), that deletes \(m\) consecutive elements from a list starting with the \(n^{th}\) element. The reduction rule for this family is:

\[
(K)[m,n] f (x_1, \ldots, x_{n-1}, x_n, \ldots, x_{n+m}, \ldots, x_p) \\
\rightarrow f (x_1, \ldots, x_{n-1}, x_{n+m}, \ldots, x_p)
\]

In this reduction rule, \(m \geq p-(n-1)\) and \(n \geq 1\). As an example, \((K)[1,3] f (1, 2, 3, 4) \equiv f (1, 2, 4)\). In terms of the existing notations and combinators,

\[
(K)[m,n] \equiv U(m+(n-1)) ((K(n))^m J^n)
\]

For example, \((K)[1,3] \equiv U(3) (K(3) J_{*}^{(3)})\). As an example, we delete the third element from \((1, 2, \ldots, n)\) with \(m=1\) and \(n=3\). Therefore, we use the \((K)[1,3]\) combinator to accomplish the task as shown by the following reduction steps:

\[
(K)[1,3] f (1, 2, \ldots, n) \\
\equiv U(1+(3-1))( (K(3))^{1} J_{*}^{(3)}) f (1, 2, \ldots, n) \\
\equiv U(3)(K(3) J_{*}^{(3)}) f (1, 2, \ldots, n) \\
\equiv K(3)(J_{*}^{(3)}) f 1 2 3 (4, \ldots, n) \\
\equiv J_{*}^{(3)} f 1 2 (4, \ldots, n) \\
\equiv J_{*}^{(3)} f 1 2 (4, \ldots, n)
\]
4.3.6 Modifying the list elements

The \((n, k)C\) combinator has been defined to modify a list by inserting new elements into it. In this section, a combinator family \((K)^{[m,n]}\) is introduced to modify a list by deleting its current elements and substituting new elements for them. The new combinator family modifies \(n\) consecutive elements of a list starting from the \(m^{th}\) element. The reduction rule for this family is:

\[
(K)^{[m,n]} f (x_1, \ldots, x_{m-1}, x_m, \ldots, x_{m+n-1}, x_{m+n}, \ldots, x_p) \rightarrow f (x_1, \ldots, x_{m-1}, x_{m+n}, \ldots, x_p)
\]

The degree of the combinator family is \(n+2\). The second argument is the list whose elements are to be modified and the third through \(m+2\) arguments are to be substituted for the modified elements. Thus, \((K)^{[3,2]} f (1, 2, 3, 4, 5) \equiv f (1, 2, 8, 9, 5)\). In terms of the existing combinators and notations,

\[
(K)^{[m,n]} \equiv U(n+(m-1))((K(m+1))^n C_{(m,n)} J_{(m+n)})
\]

For example,

\[
(K)^{[3,2]} \\
\equiv U(2+(3-1))((K(3+1))^2 C_{(3,2)} J_{(3+2)}) \\
\equiv U(4)((K(4))^2 C_{(3,2)} J_{(5)})
\]
We go through the complete reduction process to modify the third and the fourth elements of \((1,2,3,4,5,\ldots,n)\) to 8 and 9. Clearly, \(m=3\) and \(n=2\), therefore, the \((K)_{[3,2]}\) combinator is used. The reduction steps to accomplish the task are:

\[
(K)_{[3,2]} \ f (1, 2, \ldots, n)
\equiv U(4)((K(4))^2 C(3,2) J(5)) \ f (1, 2, \ldots, n) \ 8 \ 9
\equiv (K(4))^2 C(3,2) (J(5) I) \ f 1 \ 2 \ 3 \ 4 \ (5, \ldots, n) \ 8 \ 9
\equiv C(3,2) (J(5) I) \ f 1 \ 2 \ (5, \ldots, n) \ 8 \ 9
\equiv C(3) (C(4) (J(5) I)) \ f 1 \ 2 \ (5, \ldots, n) \ 8 \ 9
\equiv C(4) (J(5) I) \ f 1 \ 2 \ 8 \ (5, \ldots, n) \ 9
\equiv J(5) I \ f 1 \ 2 \ 8 \ 9 \ (5, \ldots, n)
\equiv I \ f (1, 2, 8, 9, 5, \ldots, n)
\equiv f (1, 2, 8, 9, 5, \ldots, n)
\]

### 4.3.7 Conversion of a list to multiple sublists

The combinator family, \(Z_{[n,m,k]}\), is defined to translate a list of \(n\) elements to a list of \(k\) lists of \(m\) elements each (obviously, \(n=k \cdot m\)). The reduction rule for the family is:

\[
Z_{[n,m,k]} \ f (x_1, \ldots, x_m, x_{m+1}, \ldots, x_{2m}, \ldots, x_{(k-1)m}, \ldots, x_{km})
\rightarrow f ((x_1, \ldots, x_m), (x_{m+1}, \ldots, x_{2m}), \ldots, (x_{(k-1)m+1}, \ldots, x_{km}))
\]

Hence, \(Z_{[12,4,3]} \ f (1, 2, \ldots, 12) \equiv f ((1, 2, 3, 4), (5, 6, 7, 8), (9, 10, 11, 12))\). In terms of the existing terminology and combinators,

\[
Z_{[n,m,k]} \equiv U(n) \ ((Z)_m (B (Z_k)_*))
\]
For example, \( Z_{[12,4,3]} \equiv U^{(12)}((Z)^{3}_{3} (B (Z_k)_*)) \). In order to translate \((1,2, \ldots ,9)\) to \(((1,2,3),(4,5,6),(7,8,9))\), \(Z_{[9,3,3]}\) member of the family will be used because \(n=9\) and \(m=k=3\) in this case. Following are the reduction steps when \(Z_{[9,3,3]}\) is applied to the given list:

\[
\begin{align*}
Z_{[9,3,3]} f (1,2, \ldots, 9) \\
\equiv U^{(9)}((Z)^{3}_{3} (B Z_3 I)) f (1,2, \ldots, 9) \\
\equiv (Z)^{3}_{3} (B Z_3 I) f 1 2 \ldots 9 \\
\equiv BZ_3(B(B Z_3)(B(B(BZ_3))(B Z_3 I)))(B Z_3 I)) f 1 2 \ldots 9 \\
\equiv Z_3(B(B Z_3)(B(B(BZ_3))(B Z_3 I)) f 1 2 \ldots 9 \\
\equiv B(B Z_3)(B(B(B Z_3))(B Z_3 I)) f (1,2,3) 4 \ldots 9 \\
\equiv BZ_3(B(B(B Z_3))(B Z_3 I) f) (1,2,3) 4 \ldots 9 \\
\equiv Z_3(B(B(B Z_3))(B Z_3 I) f (1,2,3)) 4 \ldots 9 \\
\equiv B(B(B Z_3))(B Z_3 I) f (1,2,3) (4,5,6) 7 8 9 \\
\equiv BZ_3(B Z_3 I f) (1,2,3) (4,5,6) 7 8 9 \\
\equiv Z_3 (B Z_3 I f (1,2,3)) (4,5,6) 7 8 9 \\
\equiv B Z_3 I f (1,2,3) (4,5,6) (7,8,9) \\
\equiv Z_3 (I f) (1,2,3) (4,5,6) (7,8,9) \\
\equiv I f ((1,2,3),(4,5,6),(7,8,9)) \\
\equiv f ((1,2,3),(4,5,6),(7,8,9))
\end{align*}
\]

### 4.3.8 Transpose of a list of lists

In this section, a combinator family, \(T^m_n\), is introduced to take the transpose of a list of \(m\) lists of \(n\) elements each. The following is the reduction rule for the
combinator:

\[ T^n_m \ f \ ((x_{11}, \ldots, x_{1n}),(x_{21}, \ldots, x_{2n}), \ldots, (x_{m1}, \ldots, x_{mn})) \]
\[ \rightarrow f \ ((x_{11}, \ldots, x_{m1}), (x_{12}, \ldots, x_{m2}), \ldots, (x_{1n}, \ldots, x_{mn})) \]

Thus,

\[ T^4_3 \ f \ ((1,2,3,4),(5,6,7,8),(9,10,11,12)) \equiv f \ ((1,5,9),(2,6,10),(3,7,11),(4,8,12)) \]

In terms of the already defined combinators and notations,

\[ T^n_m \equiv U^{(m)} ((HOZ)^n_m (B Z_n K_m)) \]

For example, \( T^4_3 \equiv U^{(3)} ((HOZ)^4_3 (B Z_4 K_3)) \). To take the transpose of a list of two lists with each list having three elements, the \( T^3_2 \) family member is used. If the list is \(((1,2,3),(4,5,6))\), the following reduction steps illustrate the semantics of this combinator family:

\[ T^3_2 \ f \ ((1,2,3),(4,5,6)) \]
\[ \equiv U^{(2)} ((HOZ)^3_2 (B Z_3 K_2)) \ f \ ((1,2,3),(4,5,6)) \]
\[ \equiv ((HOZ)^3_2 (B Z_3 K_2)) \ f \ (1,2,3) (4,5,6) \]
\[ \equiv BH_2(BO_2(BZ_2(B(B H_2)(B(B O_2)(B(B Z_2)(B(B(B H_2))(B(B(B O_2))\]
\[ (B(B(B Z_2)) (B Z_3 K_2)))))))) \ f \ (1,2,3) (4,5,6) \]
\[ \equiv H_2(BO_2(BZ_2(B(B H_2)(B(B O_2)(B(B Z_2)(B(B(B H_2))(B(B(B O_2))\]
\[ (B(B(B Z_2)) (B Z_3 K_2)))))))) \ f \ (1,2,3) (4,5,6) \]
\[ \equiv BO_2(BZ_2(B(B H_2)(B(B O_2)(B(B Z_2)(B(B(B H_2))(B(B(B O_2))\]
\[ (B(B(B Z_2)) (B Z_3 K_2)))))))) \ f \ (1,2,3) (4,5,6) \]
\[ \equiv O_2(BZ_2(B(B H_2)(B(B O_2)(B(B Z_2)(B(B(B H_2))(B(B(B O_2))\]
\[ (B Z_3 K_2)))))))) \ f \ (1,2,3) (4,5,6) \]
\[
\begin{align*}
\equiv B Z_2(B(B H_2)(B(B O_2)(B(B Z_2)(B(B(B H_2))(B(B(B O_2))(B(B(B Z_2))
& (B Z_3 K_2))))))) f 1 4 (2,3) (5,6) \\
\equiv Z_2(B(B H_2)(B(B O_2)(B(B Z_2)(B(B(B H_2))(B(B(B O_2))(B(B(B Z_2))
& (B Z_3 K_2))))))) f 1 4 (2,3) (5,6) \\
\equiv B(B H_2)(B(B O_2)(B(B Z_2)(B(B(B H_2))(B(B(B O_2))(B(B(B Z_2))
& (B Z_3 K_2)))))) f (1,4) (2,3) (5,6) \\
\equiv B(B O_2)(B(B Z_2)(B(B(B H_2))(B(B(B O_2))(B(B(B Z_2))
& (B Z_3 K_2)))))) f (1,4) (2,3) (5,6) \\
\equiv B(B O_2)(B(B Z_2)(B(B(B H_2))(B(B(B O_2))(B(B(B Z_2))
& (B Z_3 K_2)))))) f (1,4) 2 (3) 5 (6) \\
\equiv B(B H_2)(B(B O_2)(B(B(B H_2))(B(B(B O_2))(B(B(B Z_2))
& (B Z_3 K_2)))))) f (1,4) 2 (3) 5 (6) \\
\equiv B(B Z_2)(B(B(B H_2))(B(B(B O_2))(B(B(B Z_2)) (B Z_3 K_2))))
& f (1,4) 2 (3) 5 (6) \\
\equiv B(B Z_2)(B(B(B H_2))(B(B(B O_2))(B(B(B Z_2)) ) (B Z_3 K_2)))) f
& (1,4) 2 5 (3) (6) \\
\equiv Z_2(B(B(B H_2))(B(B(B O_2))(B(B(B Z_2)) ) (B Z_3 K_2)))) f
& (1,4) 2 5 (3) (6) \\
\equiv B(B(B H_2))(B(B(B O_2))(B(B(B Z_2)) ) (B Z_3 K_2)))) f (1,4) (2,5) (3) (6) \\
\equiv B(B B_2)(B(B(B O_2))(B(B(B Z_2)) ) (B Z_3 K_2)))) f (1,4) (2,5) (3) (6)
\end{align*}
\]
4.3.9 Reversing a list

Finally, a combinator, \( ^nC \), is introduced to reverse a list of \( n \) elements. The reduction rule for this family is:

\[
^nC f (x_1, \ldots, x_n) \rightarrow f (x_n, \ldots, x_1)
\]

Thus, \( ^3C I (1, 2, 3) \equiv I (3, 2, 1) \equiv (3, 2, 1) \). \( ^nC \) can also be represented as:

\[
^nC \equiv U^{(n)} (nC (B Z_n (B K))
\]

where

\[
nC \equiv C^{[n+1,n-1]}C^{[n+1,n-2]} \ldots (C^{[n+1,n-(n-1)]}) \ldots
\]
where
\[ C^{[m,n]} = C_m(C_(m-1)(C_(m-2)\cdots(C_(m-(n-1))\cdots)) \]

Hence, \[ 3C = U^{(3)}(3C(BZ_3(BK))) \]. Therefore, if the input list is \((1,2,3)\), the reversal of the list takes place with the following reduction sequence:

\[
\begin{align*}
3C_f(1,2,3) &
\equiv U^{(3)}(3C(BZ_3(BK)))(1,2,3) \\
&
\equiv 3C(BZ_3(BK))f123() \\
&
\equiv C^{[4,2]}(C^{[4,1]}(BZ_3(BK)))f123() \\
&
\equiv C^{[4]}(C^{[3]}(C^{[4]}(BZ_3(BK))))f123() \\
&
\equiv C^{[4]}(C^{[3]}(BZ_3(BK)))f132() \\
&
\equiv C^{[4]}(BZ_3(BK))f321() \\
&
\equiv BZ_3(BK)f321() \\
&
\equiv Z_3((BK)f)321() \\
&
\equiv BK(f(3,2,1)()) \\
&
\equiv K((f(3,2,1)())() \\
&
\equiv f(3,2,1)
\end{align*}
\]

4.4 Parallel List Combinators

All the list combinators that have been described so far access and manipulate the list elements sequentially. However, there are a set of list operations whose semantics require concurrent access of list elements. Examples of such operations are \textit{map}, \textit{transpose}, \textit{distribute left}, \textit{distribute right}, etc. Although implementation of the
basic and the major list combinators will itself result in a very powerful language, the
capability of the language will be enhanced if map-like parallel operators are treated
as combinators. As we shall see in the next sections, these operations distribute one
argument to elements of a list. Traditionally, this distribution is carried out over
one element of a list at a time, although list elements can potentially be accessed
simultaneously.

In this section, a set of combinators is introduced to perform parallel oper­
ations on the list structure by accessing the list elements simultaneously. These
operations are of utmost importance while manipulating arrays, lists, and vectors
in applications like database searches, artificial intelligence, and image processing.
The reduction rules and equivalent strings of already known combinators are given
for each of these combinator families. The reduction steps for each of the equivalent
strings approximately manifests the reduction semantics for the combinator family
it represents.

4.4.1 distl (DL)

The distl operator takes a list of two elements \((f,L)\) (where \(L\) is a list) as an
argument, distributes \(f\) to each element of \(L\), and returns a list of lists according to
following recursive definition written in a SASL-like language:

\[
\text{distl}(f, L) = L = () \rightarrow ()
\]

\[
\text{cons } ((f, \text{hd}(L)), \text{distl}(f, \text{tl}(L)))
\]

Here, juxtaposition represents functional application. As a combinator, distl is
denoted as \(DL_n\) where \(n\) is the number of elements in the second argument and has
the following reduction rule:

\[ DL_n f(x, (x_1, x_2, \ldots, x_n)) \rightarrow f((x, x_1), (x, x_2), \ldots, (x, x_n)) \]

Thus, \( DL_2 f ((5,6), ((1,2), (3,4))) \equiv f (((5,6), (1,2)), ((5,6), (3,4))) \). In terms of the already discussed combinators,

\[ DL_n \equiv U^{(2)}(B U^{(n)}(M_n ((Z)^2 (B Z_n ((K_1)^2)))))) \]

The following example verifies the validity of this equivalence equation:

\[
\begin{align*}
DL_2 f (1,(2,3)) &\equiv U^{(2)}(B U^{(2)}(M_2((Z)^2 (B Z_2 ((K_1)^2)))))) f (1,(2,3)) \\
&\equiv B U^{(2)}(M_2((Z)^2 (B Z_2 ((K_1)^2)))) f 1 (2,3) () \\
&\equiv U^{(2)}(M_2((Z)^2 (B Z_2 ((K_1)^2)))) f 1 (2,3) () \\
&\equiv M_2((Z)^2 (B Z_2 ((K_1)^2)))) f 1 2 3 () () \\
&\equiv (Z)^2 (B Z_2 ((K_1)^2)) f 1 2 3 () () \\
&\equiv B Z_2 ((K_1)^2) f (1,2) (1,3) () () \\
&\equiv Z_2 ((K_1)^2 f) (1,2) (1,3) () () \\
&\equiv (K_1)^2 f ((1,2) (1,3)) () () \\
&\equiv f ((1,2) (1,3))
\end{align*}
\]

4.4.2 distr (DR)

The distr operator is similar to distl operator except for the order of input list elements. Unlike distl, distr takes a two-element list whose first element is a list and returns a list according to the following definition:

\[
distr(L, f) = L = () \rightarrow ()
\]

\[
cons ((hd(L), f), distr(tl(L), f))
\]
Again, juxtaposition represents functional application. As a combinator, distr is denoted as $DR_n$ and has the following reduction rule:

$$DR_n f ((x_1, \ldots, x_n), x) \rightarrow f ((x_1, x), \ldots, (x_n, x))$$

Hence, $DR_2 f (((1, 2), (3, 4)), (5, 6)) \equiv f (((1, 2), (5, 6)), ((3, 4), (5, 6)))$. In terms of the already defined combinators,

$$DR_n \equiv U^{(2)}(U^{(n)})(C_{n+1})(N_n((Z)^2_2 (B Z_2 ((K_1)^2))))$$

The following example illustrates the semantics of the $DR_2$ combinator:

$$DR_2 f ((2, 3), 1)$$

$$= U^{(2)}(U^{(2)}(C_3)(N_2((Z)^2_2 (B Z_2 ((K_1)^2)))))) f ((2, 3), 1)$$

$$= U^{(2)}(C_3)(N_2((Z)^2_2 (B Z_2 ((K_1)^2)))) f (2, 3) 1 ()$$

$$= C_3(N_2((Z)^2_2 (B Z_2 ((K_1)^2)))) f 2 3 1 ()$$

$$= N_2((Z)^2_2 (B Z_2 ((K_1)^2)))) f 2 3 1 ()$$

$$= (Z)^2_2 (B Z_2 ((K_1)^2))) f 2 3 1 ()$$

$$= B Z_2 ((K_1)^2) f (2, 1) (3, 1) ()$$

$$= Z_2 ((K_1)^2) f (2, 1) (3, 1) ()$$

$$= (K_1)^2 f ((2, 1), (3, 1)) ()$$

$$= f ((2, 1), (3, 1))$$

### 4.4.3 Inner Product (IP)

The inner product (IP) operator takes a list of lists as input, performs the transpose of all the lists, computes the product of the elements of the resultant lists, and returns the sum of all the products. As a combinator, $IP_{m,n}$ takes
the inner product of \( m \) lists of \( n \) elements each and has the following equivalence equation:

\[
IP_{m,n} \equiv T_m^m (MAP_m (ADD_m I) MULT_n)
\]

Here, \( ADD_m \) and \( MULT_n \) are combinators which perform vector add and vector multiply pairs of \( m \) and \( n \) elements, respectively. The semantics of \( IP_{m,n} \) are specified by the following example:

\[
\begin{align*}
IP_{2,2} ((1,2),(3,4)) \\
\equiv T_2^2 (MAP_2 (ADD_2 I) MULT_2) ((1,2),(3,4)) \\
\equiv MAP_2 (ADD_2 I) MULT_2) ((1,3),(2,4)) \\
\equiv ADD_2 I ((MULT_2 (1,3)), (MULT_2 (2,4))) \\
\equiv ADD_2 I (3,8) \\
\equiv I 11 \\
\equiv 11
\end{align*}
\]

### 4.4.4 MAP

The \( MAP \) operator takes two arguments \( L \) and \( f \) (where \( L \) is a list) as input, distributes \( f \) to each element of \( L \), and returns a list that contains all the elements of \( L \) with \( f \) applied to each. The working of this operation can be specified by the following SASL script:

\[
MAP f L = L = () \rightarrow () \\
\quad cons ((f (hd(L))), MAP f (tl(L)))
\]

As a combinator, \( MAP_n \) has the following reduction rule:

\[
MAP_n f g ((x_1,x_2), (x_3,x_4), \ldots, (x_{2n-1},x_{2n})) \rightarrow \\
f ((g (x_1, x_2)), (g (x_3, x_4)), \ldots, (g (x_{2n-1}, x_{2n})))
\]
For example, $MAP_3 f + ((1,2),(3,4),(5,6)) \equiv f ((+ (1,2)), (+ (3,4)), (+ (5,6)))$.

In terms of already defined combinators,

$$MAP_n \equiv B U^{(n)} (M_n (L_n J^{(n)}_*))$$

Therefore, $MAP_3 \equiv B U^{(3)} (M_3 (L_3 J^{(3)}_*))$. The reduction process for $MAP_3$ illustrates the execution of the $MAP_n$ combinator family:

$$MAP_3 f + ((1,2),(3,4),(5,6))$$
$$\equiv B U^{(3)} (M_3 (L_3 J^{(3)}_*)) f + ((1,2),(3,4),(5,6))$$
$$\equiv U^{(3)} (M_3 (L_3 J^{(3)}_*)) f + ((1,2),(3,4),(5,6))$$
$$\equiv M_3 (L_3 J^{(3)}_*) f + (1,2)(3,4)(5,6)$$
$$\equiv L_3 J^{(3)}_* f + (1,2)(3,4)(5,6)$$
$$\equiv J^{(3)}_* f (+ (1,2))(+ (3,4))(+ (5,6))$$
$$\equiv f ((+ (1,2)),(+ (3,4)),(+ (5,6)))$$

### 4.5 An Illustrated Example

To illustrate the evaluation and usefulness of the combinators developed in the previous sections, the matrix multiply algorithm is used once again. The SASL script for the algorithm was given in Chapter 2. Again, as was done in Chapter 2, two $2 \times 2$ matrices are multiplied. Here, the equivalent combinator string and the reduction steps for the complete reduction of the string are given. The SASL script to multiply two $n \times n$ matrices (as given in Chapter 2) can be compiled into the following combinator string:

$$MM_n \equiv B T^n_m (Z_2 (DR_n (MAP_n (MAP_n I (MAP_n I IP_{m,n})) (DL_n)_*))))$$
In our example, \( n=m=2 \). The complete reduction process for the multiplication of \(((1,2),(3,4))\) and \(((1,2),(3,4))\) is given below:

\[
MM_2 \left(\begin{array}{cc}
(1,2) & (3,4) \\
(1,2) & (3,4)
\end{array} \right) \\
\equiv B T_2^2 \left( Z_2 \left( DR_2 \left( MAP_2 \left( MAP_2 I \left( MAP_2 I IP_2,2 \right) \right) \left( DL_2 \right)_* \right) \right) \right) \left(\begin{array}{cc}
(1,2) & (3,4) \\
(1,2) & (3,4)
\end{array} \right)
\]

\[
\equiv T_2^2 \left( Z_2 \left( DR_2 \left( MAP_2 \left( MAP_2 I \left( MAP_2 I IP_2,2 \right) \right) \left( DL_2 \right)_* \right) \right) \right) \left(\begin{array}{cc}
(1,2) & (3,4) \\
(1,2) & (3,4)
\end{array} \right)
\]

\[
\equiv Z_2 \left( DR_2 \left( MAP_2 \left( MAP_2 I \left( MAP_2 I IP_2,2 \right) \right) \left( DL_2 \right)_* \right) \right) \left(\begin{array}{cc}
(1,2) & (3,4) \\
(1,2) & (3,4)
\end{array} \right)
\]

\[
\equiv DR_2 \left( MAP_2 \left( MAP_2 I \left( MAP_2 I IP_2,2 \right) \right) \left( DL_2 \right)_* \right) \left(\begin{array}{cc}
(1,2) & (3,4) \\
(1,2) & (3,4)
\end{array} \right)
\]

\[
\equiv MAP_2 \left( MAP_2 I \left( MAP_2 I IP_2,2 \right) \right) \left(\begin{array}{cc}
(1,2) & (3,4) \\
(1,2) & (3,4)
\end{array} \right)
\]

\[
\equiv I \left(\begin{array}{cc}
(((1,2),(1,3)),((1,2),(2,4))) \\
(((3,4),(1,3)),((3,4),(2,4)))
\end{array} \right)
\]

\[
\equiv I \left( ((IP_{2,2} ((1,2),(1,3))), (IP_{2,2} ((1,2),(2,4)))) \right) \left(\begin{array}{cc}
(3,4) & (1,3) \\
(3,4) & (2,4)
\end{array} \right)
\]

\[
\equiv I \left( ((IP_{2,2} ((1,2),(1,3))), (IP_{2,2} ((1,2),(2,4)))) \right) \left(\begin{array}{cc}
(3,4) & (1,3) \\
(3,4) & (2,4)
\end{array} \right)
\]

\[
\equiv I \left( ((IP_{2,2} ((1,2),(1,3))), (IP_{2,2} ((1,2),(2,4)))) \right) \left(\begin{array}{cc}
(3,4) & (1,3) \\
(3,4) & (2,4)
\end{array} \right)
\]

\[
\equiv I \left( ((7,10),(15,22)) \right) \equiv ((7,10),(15,22))
\]
In this example, parallelism is invoked wherever possible. For example, two \((DL_2)^*\) and four \(IP_{2,2}\) reductions are performed in parallel in reduction steps six and ten, respectively. Note that whenever distribute- and map-like functions are encountered and the element that is distributed is a combinator, the reduction of both the distribute combinator and the next combinator in the combinatory sequence can be performed simultaneously on a multiprocessing system. Also, not only the distribute- and map-like combinators are parallel, their reduction produces a code that has sub-expressions which can be reduced in parallel. The advantages of this approach are two-fold:

1. Multiple subexpressions can be reduced in parallel.

2. There is no data dependency between separate expressions and while these expressions are being reduced in parallel, the next reduction in the sequence can be performed simultaneously because the combinator that is to be reduced next in the sequence has its argument in appropriate form.

Another point to note is that the reduction of the next combinator appends a new element at the head of the list, thus preserving the sharing property of recursively-defined lists.

In the next chapter, a new list representation is proposed to efficiently support the proposed list combinators.
5 LIST REPRESENTATION

5.1 Introduction

As mentioned previously, the semantics of the combinators developed in Chapter 4 demand a list structure that is different from the conventionally-defined recursive structure. In this section, a new list structure is proposed that accommodates the semantics of the list combinators very nicely. The new list implementation is a modified vector-coded representation, hereafter referred to as the concurrent list, that maps well onto the semantics of the list combinators. The concurrent list supports inexpensive indexing, insertion, and deletion, and allows multiple processes to access different parts of the same list simultaneously by using a simple locking mechanism. Some examples are given to illustrate the mapping of the list combinators onto the concurrent list.

5.2 The Concurrent List

The concurrent list is an enhancement of the conc-representation which is a vector-coded representation [15]. In a conc-representation, a list is represented as vectors stored in contiguous memory locations. Each vector is denoted by the 2-tuple (N,A), where N is the number of elements in the vector and A is the address of the first element of the vector. Special tuples, called conc-cells, have two of their
elements as pointers to other conc-cells or tuples. The conc-cells are used to do
list concatenation without destroying the original structure of the first list. This
is accomplished by having one pointer of a conc-cell point to the last element of
the first list and the second pointer of the conc-cell point to the first element of
the second list. The major advantage of performing concatenation in this way is
that the sharing of the list is not destroyed because none of the cdr pointers are
modified.

Each vector cell is composed of three fields: the data field, the control field,
and the cdr field. The data field holds a list element or a pointer. The control field
determines whether the current cell is used and if so, whether it contains a data
element or a cdr pointer to the next element. The cdr field contains a pointer to the
next list element. By default, however, the next element resides in the next cell.

5.2.1 Logical representation of the concurrent list

A concurrent list is represented as an ordered set of vectors. Each vector con­
tains a number of list elements stored in contiguous memory locations and is rep­
resented by a 6-tuple:

\[(G, L, N_v, A, D, NEXT)\]

where G is the garbage bit, L is a lock bit, \(N_v\) is the number of cells in the vector
(i.e., size of the vector), A is the address of the first (or last) element of the vector,
D is the direction bit (0 for forward direction and 1 for backward direction), and
NEXT is the address of the next tuple. The NEXT field is NIL for the last tuple.
A special tuple, called the head-tuple, is used to represent the complete list. The
head-tuple is a 5-tuple:

\[(G, N_l, A_1, A_2, F)\]

where \(G\) is the garbage bit, \(N_l\) is the number of elements in the list (i.e., size of the list), \(A_1\) and \(A_2\) are addresses of the first and the last tuples, and \(F\) is a bit that tells which of the two addresses, \(A_1\) and \(A_2\), is the address of the first tuple.

\[\text{5.2.1.1 The vector-cell structure} \quad \text{A cell in a vector is composed of three fields: the data field, the status field, and the cdr abnormal field. The data field contains a list element which can contain an atom (character, integer, real, etc.) or a pointer to a tuple–which accommodates list insertions and multi-level lists. The cdr abnormal field contains offset to the next vector element. By default, however, the next vector element resides in the next cell. The cdr abnormal for the last element of a vector is NIL. The status field contains information about three things: (i) the status of the cell (used/unused), (ii) the type of data in the data field of the cell, and (iii) the information about the cdr of this element. Three bits are used to encode the status field information: one bit to indicate whether the current cell is used, one bit to indicate whether the element in the data field of the cell is a list element or a pointer to a tuple, and one bit to indicate if the cdr of this element is normal (i.e., the next vector element resides in the next vector) or abnormal (i.e., the cdr abnormal field contains the offset to the next vector element).

To access a vector element, a process acquires of the appropriate tuple and checks if the lock bit is set. If it is set, the process waits. Otherwise, the process sets the lock bit, accesses the vector element it needs, and decodes the control field to determine if: (i) the current cell is garbage (unused), (ii) the element is an atom}
or a pointer to a tuple, (iii) the cdr of this cell is normal or abnormal. If the
current cell is garbage, the next non-garbage cell contains the needed element. If
the cdr bit is set (i.e., the cdr is abnormal), the offset to the address of the next
element is obtained from the cdr abnormal field. Figure 5.1 shows the concurrent-
list representation for (1,2,3, ... ,100) with a vector size (referred to as ‘size’ in the
discussion to follow) of 10.

5.2.2 Memory management

The user memory is divided into two sections: the tuple memory and the vector
memory. The tuple memory is further subdivided into two parts: the head-tuple
memory and the normal-tuple memory. The head-tuple memory contains a free
list of head-tuples and the normal-tuple memory contains a free list of tuples. The
vector memory contains a free list of vectors. All tuples have the same size. However,
the size of a vector can be fixed or variable, and there are trade-offs associated with
each scheme.

5.2.2.1 Fixed-size vectors In this scheme, all vectors have the same size.
The advantage is that memory management becomes easy. However, there are two
problems associated with this scheme. First, if vector size is too large, there is a
potential for wasted space. Second, if the vector size is too small, the percentage
overhead increases because too many tuples are used. The trade-off, therefore, is
the waste of memory space due to unused vector cells versus the waste of memory
space due to excessive use of tuples.
5.2.2.2 Variable-size vectors In this scheme, vectors have variable sizes depending on the application at hand. Although the use of variable-size vectors is more space-efficient in some cases, the memory management becomes complex. We can either adopt a scheme where free lists are maintained for different sized vectors or we can have a mechanism equivalent to a segmented memory system where a vector is searched for and allocated at run-time to meet the current requirement. The former scheme becomes expensive if the system runs out of vectors of some particular size. If the available vectors are of sizes greater than the required size, there is potential for wasted vector cells and if the sizes of the available vectors are smaller than the required size, an appropriate sized vector has to be constructed at run-time. For the latter mechanism, external fragmentation can become a problem.

5.3 Examples

In this section, examples are given for some representative list operations. The set of these operations has been chosen carefully enough to encompass the execution semantics of most of the list combinators described in this dissertation. To enhance clarity, algorithms as well as pictorial representations are given for all the operations that are discussed. In the algorithmic description, \( N_l \) refers to the number of elements in the list and \( N_v \) denotes the number of elements in a vector. The list structure shown in Figure 5.1 will be taken as the input list for all the examples presented in this section.
5.3.1 List reversal

The list reversal considered in this section is the standard one-level list reversal and corresponds to the $^nC$ combinator. However, the algorithm can be easily extended to accommodate multi-level list reversal. The following algorithm performs the reverse operation on a list:

```plaintext
REVERSE(L)
begin
  Invert the F bit of the head-tuple.
  Invert the D bits of all the tuples.
end.
```

Figure 5.2 shows the input list after having performed the REVERSE operation on it. Note that after the reversal has taken place, $A_2$ and $A_1$ represent the addresses of the first and the last tuples, respectively. $(A+\text{size})$ represents the address of the first vector element.

5.3.2 List indexing

The index operation returns the $i^{th}$ element for some $i$. If $i$ is greater than the list size, an error message is returned. This operation exactly maps the semantics of the $D(m)$ combinator described in the previous chapter. The following algorithm illustrates the operations that are performed on the input list:
Figure 5.1: Concurrent list representation of (1,2, ... ,100)
INDEX\((L, i)\)
begin
    if \(N_l < i\) then ERROR;
    otherwise begin
    Find the tuple for the vector that contains the \(i^{th}\) element by scanning the \(N_v\) field of every tuple (starting from the first tuple) one at a time.
    Return the first used (non-garbage) element \(\geq (i \ \text{REM} \ n)\) where \(n\) is the sum of \(N_v\)'s for all previous tuples.
    end.
end.

Note that more than one list elements can be accessed simultaneously by using the lock bit if the accessed elements reside in different vectors.

5.3.3 List insertion

The insertion operation described in this section inserts \(i\) elements into a list starting from the \(j^{th}\) position. The operation exactly maps onto the semantics of the \([n,k]\) C combinator described in Chapter 4. An algorithmic description of the operation follows:
Figure 5.2: Concurrent list representation of (100, 99, ..., 1)
INSERT(L,i,j)
begin
    Acquire of the appropriate tuple by checking the \( N_v \) fields of the tuples one at a time (starting from the first tuple).
    if \( j \) is the 1\(^{st}\) or last element of a vector
    then begin
        Allocate appropriate number of vectors and tuples and fill them with appropriate values.
        Update \( N_l \) and \( A_1 \) (or \( A_2 \)) fields of the head-tuple.
    end.
    if \( j \) is not the 1\(^{st}\) or last element position of a vector
    then begin
        Allocate appropriate number of vectors and tuples and fill them appropriately.
        Update the \textit{cdr abnormal} field of \((j - 1\text{st})\) element.
        Update the \( N_v \) field of the tuple that represents the updated vector.
        Update \( N \) and \( A_1 \) (or \( A_2 \)) fields of head-tuple.
    end.
end.

Figure 5.3 shows the insertion of 3, 4, and 5 starting from the 17\(^{th}\) element. The INSERT operation can be applied to different portions of a list provided the insertions take place in different vectors.

5.3.4 List deletion

The delete operation described below deletes \( i \) elements from a list starting from the \( j^{th} \) element. This operation exactly maps the semantics of the \( K_{[m,n]} \) combinator described in the previous chapter. The following algorithm describes the operations that need to be performed to accomplish this:
Figure 5.3: Concurrent list representation after the insertion of 3, 4, and 5 at the 17th position
DELETE\((L, i, j)\)
begin
if \((i < N_v)\) and \(j\) is the 1st. element of some vector
then begin
Delete appropriate number of elements from the list.
Set the control flags for the deleted cells appropriately.
Update the \(N_v\) field in the corresponding tuple.
Update the \(N\) field in the head-tuple.
end.
else if \((i \geq N_v)\) and \(j\) is the 1st. element of some vector
then begin
Delete appropriate number of elements from the list.
Set the control flags for the deleted cells appropriately.
deallocate the vectors and the tuples that become free.
Update \(N_t\) and \(A_1\) (or \(A_2\)) fields of the head-tuple.
Update the \(N_v\) field of the last tuple affected by
the deletion process.
end.
else if \((i \geq N_v)\) and \(j\) is not the last element of some vector
then begin
Delete appropriate number of elements from the list.
Set the control bits of the deleted cells appropriately.
Update the \(cdr\) field of the \((j - 1)st\) element
appropriately.
Update \(N_t\) to an appropriate value.
Update the \(N_v\) fields of the affected tuples to
appropriate values.
If a vector is completely deleted in the process,
deallocate the vector and its associated tuple.
Update the NEXT field of the tuple that represents
the vector containing the \(j^{th}\) element.
end.
end.

Figure 5.4 shows the input list after having deleted 7, 8, 9, 10, 11, 12, and 13
from the list. The DELETE operation can be used to delete different portions of a
list simultaneously if the elements to be deleted reside in different vectors.
Figure 5.4: Concurrent list representation after the deletion of 7, ..., 13 from the list
6 SUMMARY, CONCLUSIONS, AND FUTURE RESEARCH

6.1 Summary and Conclusions

This dissertation has introduced a set of combinator families to efficiently perform list-related operations, such as constructing and destroying a list. Most of these combinators reshape the list so that the resulting structure can be manipulated to perform concurrent operations on elements of the original list. The proofs of correctness of these combinators are given, with the help of some additional terminology, in terms of the already known/defined combinators. A new list structure is then proposed which accommodates the semantics of these combinators very nicely. The semantics of the new list combinators and the architecture of the concurrent list combine to help remove the functional bottleneck.

To measure the effectiveness of the proposed combinators, the matrix multiply algorithm was analyzed for a number of matrix sizes. The graph in Figure 6.1 shows the correspondence between the size of the matrices and the number of reductions performed for every size. The actual number of reduction steps is not as important in my analysis as are shapes of the graphs. Note that for Turner's combinatory model, the number of reduction steps increases exponentially for an increase in the size of matrices. The same relationship would be true for any other recursive abstraction method, such as Abdali's. The number of reduction steps performed
Figure 6.1: Correspondence between the matrix size and the reduction steps for matrix multiply

Turner's Combinators

Proposed Combinators = 15n² + 74n + 38
and the complexity of the combinatory code in the case of Abdali's abstraction method will be even more if the combinatory code is reduced sequentially because:

1. Abdali's set of combinators is much smaller than Turner's and, therefore, results in longer combinatory code in certain situations. For matrix multiply, the code for the most heavily used function, map, is longer for Abdali's algorithm as opposed to Turner's.

2. An Abdali combinator is more complex than a Turner combinator.

However, the number of effective reduction steps will be much lesser if parallelism is exploited during the reduction of Abdali's combinatory code. On the other hand, if the list combinators developed here are used, the second-order polynomial relationship is achieved between the matrix size and the number of reductions. Again, the number of reduction steps is not too important here because it depends upon the level (granularity) of the list combinators used. Thus, the basic list combinators (Section 4.2) are at a lower level (i.e., have lower granularity) than the major list combinators (Section 4.3). In my analysis, the basic list combinators are assumed to be the fundamental set of combinators and the reduction of an \(X_n\) combinator is considered \(n\) times as expensive as the reduction of the \(X\) combinator. For example, the reduction of \(C_{(n)}\) is considered to be equivalent to \(n\) \(C\)-combinator reductions. The assumption of basic list combinators is fairly relaxed because even the major list combinators could have been assumed to be the fundamental set of combinators as the structure of the concurrent list accommodates a large number of major list combinators very nicely. The second-order polynomial as opposed to an exponential relationship between the number of reduction steps and the matrix size is achieved
by using the new list combinators because:

1. The *overhead combinatory code* is reduced as the conditionals are executed only once throughout the life-time of a list operation.

2. The parallelism is exploited whenever it is possible as the reduction of list combinators results in independent expressions.

3. The vector add and vector multiply operations are allowed.

One limitation of the proposed list combinators is that they are applicable to finite lists (at least in their current form). However, this limitation can be avoided if an incremental abstraction algorithm is used for code generation. The combinatory explosion can be avoided if abstraction is done on a number of list elements (as is done for the proposed list combinators) instead of one list element at a time (as is done by Turner for his U and P combinators). A pragmatic issue is that the programmer needs to specify the size of a list at the time of its use.

It is concluded that the proposed list combinators and corresponding list representation establish a very efficient implementation of functional languages for both sequential and parallel architectures.

### 6.2 Future Research

In order to further the work described in this dissertation, research can be conducted on the following topics:

- Optimization of the proposed combinators.
• Development of abstraction rules for the proposed list combinators. This is necessary in order to compile a functional language into a combinatory code that includes these combinators.

• Space and time complexity of the proposed combinators.

• Completeness of the proposed combinators.

• Design of a memory architecture to efficiently support the concurrent list structure.

• Performance evaluation of the proposed combinators.
7 ACKNOWLEDGEMENTS

It is a pleasure for me to express my appreciation to the people who influenced the completion of this work. My deepest gratitude goes to Dr. James A. Davis and Dr. Douglas W. Jacobson for providing financial support throughout my Ph.D. program. I sincerely thank Dr. James A. Davis and Dr. Arthur V. Pohm, my co-major professors, for their perceptive, skillful, and tolerant directions throughout this work, with special thanks to Dr. Davis for introducing me to lambda calculus and combinators. I also thank Dr. Terry Smay, Dr. Arthur Oldehoeft, and Dr. Chester Comstock for serving on my committee.

I thankfully acknowledge the help I received from my friends, especially Bill Armstrong for carefully reading my dissertation and offering some important suggestions, Suk-Jea Hahn for having key discussions with me on the implementation of functional languages that ultimately led the way to this research, Muhammad Shafiq and Gwenna Jacobson for providing a great deal of emotional support, and Sunil Gaitonde for always reminding me how miserable the life of a Ph.D. student is. This is one of the first dissertations/theses written using \LaTeX{} on ISU Computation Center’s NAS AS/9160 computer. This endeavor, which at times was an adventure, would not have been possible without the generous help of George Christensen. Thanks George.
On the more personal side, my very sincere appreciation goes to my family, especially to my parents for guiding and helping me throughout my life, for teaching me the right values, and for all their prayers. Thank you Abbu Jee and Ammee Jee. I thank Masood, my elder brother, without whose financial support I perhaps could never have made it to this country. And I thank Aqeel, my younger brother, for being so understanding and carefully proofreading various chapters of this dissertation.
8 BIBLIOGRAPHY


