1987

Combinators as control mechanisms in multiprocessor systems

Deborah Lee Knox
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/rtd

Part of the Computer Sciences Commons

Recommended Citation
Knox, Deborah Lee, "Combinators as control mechanisms in multiprocessor systems" (1987). Retrospective Theses and Dissertations. 8553.
http://lib.dr.iastate.edu/rtd/8553

This Dissertation is brought to you for free and open access by Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
INFORMATION TO USERS

While the most advanced technology has been used to photograph and reproduce this manuscript, the quality of the reproduction is heavily dependent upon the quality of the material submitted. For example:

- Manuscript pages may have indistinct print. In such cases, the best available copy has been filmed.

- Manuscripts may not always be complete. In such cases, a note will indicate that it is not possible to obtain missing pages.

- Copyrighted material may have been removed from the manuscript. In such cases, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, and charts) are photographed 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 also filmed as one exposure and is available, for an additional charge, as a standard 35mm slide or as a 17"x 23" black and white photographic print.

Most photographs reproduce acceptably on positive microfilm or microfiche but lack the clarity on xerographic copies made from the microfilm. For an additional charge, 35mm slides of 6"x 9" black and white photographic prints are available for any photographs or illustrations that cannot be reproduced satisfactorily by xerography.
Knox, Deborah Lee

COMBINATORS AS CONTROL MECHANISMS IN MULTIPROCESSOR SYSTEMS

Iowa State University Ph.D. 1987

University Microfilms International 300 N. Zeeb Road, Ann Arbor, MI 48106
PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark ✓.

1. Glossy photographs or pages
2. Colored illustrations, paper or print
3. Photographs with dark background
4. Illustrations are poor copy
5. Pages with black marks, not original copy
6. Print shows through as there is text on both sides of page
7. Indistinct, broken or small print on several pages ✓
8. Print exceeds margin requirements
9. Tightly bound copy with print lost in spine
10. Computer printout pages with indistinct print
11. Page(s) lacking when material received, and not available from school or author.
12. Page(s) seem to be missing in numbering only as text follows.
13. Two pages numbered. Text follows.
14. Curling and wrinkled pages
15. Dissertation contains pages with print at a slant, filmed as received
16. Other

University
Microfilms
International
Combinators as control mechanisms in multiprocessor systems

by

Deborah Lee Knox

A Dissertation Submitted to the Graduate Faculty in Partial Fulfillment of the Requirements for the Degree of DOCTOR OF PHILOSOPHY

Major: Computer Science

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
1987
# TABLE OF CONTENTS

1. INTRODUCTION .................................................................................................................. 1
   1.1 Introduction .................................................................................................................. 1
   1.2 Problem Statement ........................................................................................................ 3
   1.3 Thesis Organization ..................................................................................................... 3

2. REVIEW OF BACKGROUND MATERIAL ......................................................................... 5
   2.1 Introduction .................................................................................................................. 5
   2.2 Literature Review ......................................................................................................... 5
   2.3 Lambda Calculus ........................................................................................................... 12
      2.3.1 Functional Languages Compared to Imperative Languages ......................... 12
      2.3.2 Functional Languages and the Lambda Calculus .......................................... 12
      2.3.3 Applicative Expressions ..................................................................................... 13
      2.3.4 Useful Properties of the Lambda Calculus ...................................................... 13
      2.3.5 Correspondence Between a Functional Language and the Lambda Calculus ... 16
   2.4 Combinators .................................................................................................................. 17
      2.4.1 Useful Properties of the Combinator Model ................................................... 18
      2.4.2 Correspondence of the Lambda Calculus and Combinators ......................... 18
      2.4.3 The S-K Reduction Machine ............................................................................. 20

3. A METHODOLOGY TO GENERATE COMPUTER ARCHITECTURES .................................... 22
   3.1 Abdali's Abstraction Algorithm .................................................................................... 22
      3.1.1 Demonstration of Abdali's Abstraction Method ............................................. 25
      3.1.2 Comparison of Abdali's Method and Turner's Method .................................... 27
   3.2 Introduction of New Combinators ............................................................................... 30
      3.2.1 Revisiting Examples Using New Abstraction Rules ....................................... 33
   3.3 Optimization of the New Abstraction Method ........................................................... 37
      3.3.1 Evaluation of the Optimized Abstraction Method ........................................... 40
   3.4 Another Optimization .................................................................................................. 42
      3.4.1 More Examples ................................................................................................... 45
   3.5 Review of a Fixed-Program Machine ......................................................................... 49
   3.6 Modification of the Evaluator ...................................................................................... 53
      3.6.1 Evaluation of Examples Using the Modified Evaluator ................................ 54
   3.7 Multiprocessor Support ............................................................................................... 65
      3.7.1 Allocation of Processors .................................................................................... 65
      3.7.2 Synchronization of Execution .......................................................................... 68
      3.7.3 Deallocation of Processors ............................................................................... 69

4. IMPLEMENTATION NOTES ................................................................................................ 71
   4.1 Machine Instructions ..................................................................................................... 71
   4.2 Multiprocessor Support ............................................................................................... 73
   4.3 Management of a Finite Processor System .................................................................. 75
   4.4 System Requirements .................................................................................................. 76
1

1. INTRODUCTION

1.1 Introduction

Interest in functional programming languages and their support has increased rapidly in recent years. Although the elegance of functional languages has long been recognized, such languages have not, in general, been in wide use. One reason for this is that functional language implementations have not proven to be competitive with imperative language implementations. However, there are signs this situation is starting to change. Researchers are rediscovering that functional languages provide an inherent parallelism that is not present in sequential languages. If this parallelism could be taken advantage of, then the functional approach would be competitive with the traditional imperative approach. More importantly, this inherent parallelism would map quite nicely to multiprocessor evaluation.

One of the problems with capitalizing on the inherent parallelism found in functional languages is how to capture it. Most research being conducted involves studying a program to determine where the parallelism exists, and then deciding how to decompose the program to best take advantage of this parallelism. This decomposition of the program involves generating subprograms that "fit" the given architecture.

It is our belief that a very viable way to capture the parallelism inherent in functional programs is at the language level itself. In particular, we feel the language, via its semantic definition, should directly dictate the computer architecture upon which it is to run. It is this belief that has been the guiding force behind our work.

Our work in this area began after studying a method in which a language dictates its supporting architecture in a formal way [21,22]. In Wand's approach to developing a machine, he first analyzed the denotational semantics of the language to be supported. The
denotational semantics define very precisely the meaning of the constructs of the language. Using this definition, he redefined the language in "operational" terms. The resulting operational definition provided his basis for developing a computer to support the language. In this sense, then, the language definition itself defines the requirements for the computer. Wand shows how he analyzed the denotational semantics for a programming language in order to develop a translator for the language, and then proceeds to map the translator definitions into architectural features. He presents a simple addition expression language, derives a translator for the language, and shows that the underlying architecture "derived" from the language definition is a stack machine.

We have been unable, thus far, to utilize Wand's approach. We still believe in the concept of deriving an architecture from the language it is intended to support. We also believe that the way to do this is from the denotational semantics of the language.

We have broken the problem down and have tackled a portion of it. Our approach is to use the denotational definition of a language and to translate this definition into combinator expressions [4]. These combinator expressions have well-defined rules for interpretation associated with them. This means that the combinator based definition of the language is operational in nature, i.e., that we can use the rules as machine instructions. As we develop the machine instructions, we are beginning to specify a supporting architecture.

In the process of deriving the architecture to support our language, we found that the combinators are dual-purposed. Not only do they serve as machine instructions for our underlying evaluator, but they provide a built-in mechanism to decompose our programs. We have found that combinators can, in fact, be used as control mechanisms for parallel evaluation. In particular, the inherent parallelism of functional languages can be captured via the natural decomposition capabilities of combinators. We do not have to struggle to break apart our programs in order to take advantage of any parallelism that may be built into the function. The combinators manage the decomposition of our program and allocate
program pieces to processors for (presumably parallel) evaluation.

1.2 Problem Statement

Our belief is that computer architectures should be derived from the programming languages they are intended to support. Furthermore, we contend that the inherent parallelism of functional programming languages should automatically be supported by such a derived architecture.

Given a language, the semantics of that language can be defined denotationally. This definition is then converted into an equivalent definition that is operational in nature. Such a definition is then translated into machine constructs that represent the underlying supporting architecture.

Our approach to solving this problem focuses on the semantics of the language. We start with a language definition given denotationally in terms of the lambda calculus [2]. This definition is then mapped into combinator expressions. Since the parallelism is inherent to the semantics of the language, it should be reflected in the definitions of the combinators as well. We plan to capture this parallelism by mapping the combinators onto an underlying implementation consisting of multiple processors. The combinators not only dictate the underlying architecture needed for support, but also direct the allocation of parallel evaluation of the combinator expressions.

1.3 Thesis Organization

The remainder of this thesis describes our method to tailor (adjust) a multiprocessor system to utilize the inherent parallelism in programs during evaluation.

In Chapter 2, we review the pertinent literature, and provide background information on the theoretical foundations of lambda calculus and combinatorial logic. We introduce a
basic functional language and demonstrate how combinators can be used to represent programs of this functional language.

In Chapter 3, we introduce a newly developed set of combinators and discuss their equivalence to traditional sets of combinators. We present some comparisons among these various sets of combinators. Our new combinators are shown to be appropriate for sequential evaluation of programs, but more importantly are shown to have the capability to direct the decomposition of program code to allow for parallel execution on a multiprocessor system.

In Chapter 4, we provide supporting information on how our system is implemented. We discuss the machine instructions of our evaluator and various supporting functions required to handle the multiprocessing features. We present system requirements for both the infinite and finite processor simulations. The chapter concludes with a presentation of some examples evaluated by our multiprocessing scheme.

In Chapter 5, we analyze the performance of our multiprocessor system. We discuss some traditional performance measurements and present those measures of importance to our implementation. Based on these measurements, an analysis of an example evaluation by our multiprocessor system is conducted. The chapter concludes with a presentation of some system performance results.

Finally, in Chapter 6, we summarize the work presented and outline future research directions.
2. REVIEW OF BACKGROUND MATERIAL

2.1 Introduction

We are interested in the support of functional languages, in particular in the underlying architecture used to support the execution of programs. We are proposing to derive an architecture from the definition of a language. We base our approach on a functional language because of certain key properties of this class of languages.

Functional languages provide a high degree of expressive power. This expressiveness yields shorter code that is easier to read. More importantly, functional languages have the property of referential transparency. This property provides the potential for parallel evaluation of programs. Since the language is free from assignment statements, it is free from side-effects. This means that when evaluating an expression, subexpressions do not interfere with one another and may be evaluated in any order — even in parallel.

Our approach to supporting a functional language is to map the language through a series of transitions. This thesis demonstrates each stage of the mapping. The stages of transition are depicted below.

functional language $\rightarrow$ expression $\rightarrow$ combinator expression $\rightarrow$ machine code

The combinators of the language define the architecture features necessary to support our language, thus the language dictates the support it requires. After some brief background information, each of these steps of transition are examined in detail.

2.2 Literature Review

Functional languages are based on a mathematical model known as the lambda calculus [2]. It provides a formal method to study functions and function application. We are not interested in studying the lambda calculus, but want to use the resulting
model to express our functional language. Stoye et al. [18], Gordon [5], and others, have shown that a language can be formally defined in terms of the lambda calculus.

The denotational semantics of a language provide us with a framework to compare it against other languages. These formal semantics also allow us to establish formal proofs of correctness of individual programs. More important to our work is that the definition of a language provides the basis for a method of automatically generating an implementation of the language. It is this last feature that we are most interested in pursuing. Wand has used this capability in deriving a combinator machine, as we introduced in Chapter 1. Schmidt, in his work on the semantics of programming languages, has suggested that semantic definitions be used to "tune" computer architectures [16,17].

Functional languages can also be translated into a form from which all bound variables are removed. This representation is based on the mathematical model of combinatory logic. The roles of the eliminated bound variables are expressed by primitive operators, called combinators, and a mechanism for applying a function to its argument. Turner [19] presents a method for translating lambda calculus programs into combinator notation. Before proceeding with Turner's results, we first review some of the features of combinators.

Combinators were introduced by Schönfinkel in the early 1920s. They were based on the same area of mathematical logic that produced the lambda calculus. A combinator is an expression that is equivalent to a λ-expression that has no free variables. We are interested in using combinators because of this elimination of variables. The significance of not having variables in the code is that there is no need to maintain an environment. It is well-established that the maintenance of an environment, and subsequent variable look-up is very expensive in traditional implementations of von-Neumann languages. Another benefit of using a combinator model to represent programs is the simplicity of the resulting combinator machine to execute the code. This machine's instruction set is comprised of the...
translation rules of the combinators.

Using the combinator approach, Turner presents a method for the compilation of a high level language. He presents a method for eliminating the bound variables in an expression in order to produce combinator code. He also develops a machine to execute this code, known as the S-K Reduction Machine. By progressively reducing the combinator code, the expression is transformed to a number, or some other final value. This transformation method is unconventional. In the traditional "fixed program" machine, the code is not modified once it has been compiled. The importance of the transformations in Turner's machine is that expressions are replaced by mathematically equivalent ones.

The reduction rules in Turner's machine follow a policy of normal order reduction. This is simple to follow, but also has the advantage of being known to terminate (whenever termination is possible). This is important since it allows the machine to support non-strict functions.

Turner made some comparisons of his S-K Reduction Machine against an SECD machine. (The SECD machine, introduced by Landin [15], has become the standard approach in implementing functional programming languages.) Favorable results for the S-K machine were gathered when comparing the size of the required code and the speed of execution (i.e., the number of steps required to complete execution). Turner also points out that the S-K machine is actually more powerful than the SECD machine since the S-K machine follows a normal order reduction, while the SECD machine is an applicative machine. Modifying the SECD machine to handle unevaluated parameters (i.e. normal order reduction) appeared to slow the machine by an order of magnitude.

Jones [11] also investigated the efficiency in using combinator code (as Turner proposed) compared to implementing a lambda calculus reducer. He studied three reducers: a normal order lambda reducer, an applicative order lambda reducer, and a combinator reducer. He discovered that the combinator reducer outperformed both
lambda reducers. Among other conclusions, his results point out the high cost of the environment lookup for the lambda reducers. He concluded that combinatorial implementations of functional languages were competitive with other alternatives.

There are two other important features of Turner's machine. It requires the evaluation of a common subexpression only once, no matter how many times the subexpression is included in a program. A second benefit is that it allows a programmer to introduce abstractions into his program without any penalty in execution. The first time an abstraction is used, it is replaced by the necessary expansion. These benefits are not unique to the S-K machine, but occur because the machine is a reduction machine. However, even if the machine is compared against a lambda calculus reduction machine (that performs \(\beta\)-reductions), the S-K machine outperforms the latter. This is because the use of combinators allows for very simple reductions rules which can be performed faster than \(\beta\)-reductions.

Turner presented another algorithm for the abstraction process which handles special cases in a more efficient manner than his original method [20]. The algorithm keeps the number of combinators constant, therefore keeps the machine instructions at a small number. Abdali proposes an alternative algorithm that also handles the use of multi-variable abstraction, but uses an infinite number of combinators that need to be implemented at the machine level [1]. We will provide a detailed study of Abdali's approach later in this thesis.

Other researchers have found Turner's work to be a strong foundation to build upon. Whereas Turner implemented the S-K Reduction Machine in software, Clarke et al. [3] investigated how Turner's ideas could be implemented in hardware. The SKIM (The S, K, I Reduction Machine) implementation views the combinator rules as machine code and implements a fixed program in microcode to execute the combinator sequences of user programs. The SKIM investigators were pleased with the outcome of their project. The
simplicity of the combinators lead to a equally simple, yet fast, hardware design.

Favorable results with the SKIM project led to a successor implementation. The SKIM II project recognized improvements on microcoding, methods, and algorithms [18]. SKIM II provides the environment for software experimentation in the methods for combinator reductions. In particular, the project team is investigating the compilation of functional programs into microcode. Various optimizations to Turner's methods are presented, although the authors point out that these modifications are invisible to the high level programmer.

The SKIM II project should be able to easily incorporate the use of multiple processors and experience increased performance. It is also pointed out that current methods for combinator generation are far from satisfactory, the project should be able to make use of any improvement in that area.

Another area of investigation into implementing functional languages is that of compiling the functional programs. For conventional languages, compiled code generally executes faster than interpreted code. Jones and Muchnick present an approach for translating combinator code into fixed-program code [10]. Their method translates combinator code into stack machine code. This compiled code is then evaluated. Their evaluator operates by reducing a given combinator to its head normal form. This form is required by an extension to their algorithm to perform call-by-need. If normal order (call-by-name) evaluation is followed, the combinator expression can be reduced to a value. The call-by-need implementation allows some improvement in execution by generating pointers to common code so that it needs to be executed only once, regardless of the number of times it is called.

Jones and Muchnick indicate that their methods are comparable to the S-K Reduction Machine and the SKIM approaches. They also suggest further improving the execution by adapting the code for execution by a multiprocessor system. It became apparent to us that
this was exactly the approach we would take.

In all the work on combinators, a common theme to emerge is the need to find a set of combinators that is optimal in expressive power. In addition to the work of Turner and Abdali in finding optimal combinators, Hughes [8a] explores "super-combinators". Pointing out several problems with Turner's basic approach, including slow compilation due to the optimization rules and the numerous passes over the code as each variable is abstracted, Hughes proceeds to describe an approach to overcome these problems. His approach introduces super-combinators, which are a generalization of the class of combinators. Any \( \lambda \)-expression that contains no free variables and has a body that is an applicative form is a combinator. (An applicative form consists of variables and constants joined by application.) Combinators are considered to be constants, so a combinator can be built from other combinators. This leads to an infinite number of combinators.

A compiler would not need to contain definitions of these infinite number of combinators, but must be able to generate the definitions for the combinators it uses. Each program would have a unique set of combinators compiled. The lambda calculus is considered to be the canonical programming language. Hughes chose the language of constant applicative forms (caf's) as the graph-reduction machine code. He points out that Turner's conversion from lambda calculus programs into combinators achieves full laziness by breaking down the computation into very small, independent steps. The machine code is far removed from the program code, making it very difficult to debug. Hughes presents a new method that increases the size of the granularity of the steps of computation.

Refining the process of optimizing the granularity of the combinators, Hudak and Goldberg [6,7] introduced "serial combinators". They present a method for translating a functional program into serial combinators suitable for execution on a multiprocessor system that employs no shared memory. The serial combinators are intended to improve
on the notion of the super-combinator by making them larger to retain locality and improve efficiency, but also by ensuring that no parallelism is lost. Compared to Turner's method, which requires \( n \) abstractions for the translation process and \( n \) reductions to execute an expression with a free variable at lexical depth \( n \), the serial combinator method requires constant overhead to translate from \( \lambda \)-expressions into combinator notation.

Hudak's and Goldberg's intent is to provide a general-purpose system on which a user could write and debug a functional program which is then run on a parallel machine for improved performance. The parallel machine has no special need for communications or synchronization primitives or for special parallel constructs. This is in contrast to the work on the Rediflow project [12,13], where the programmer defines the granularity by source level function definitions.

Another graph reduction machine is the G-machine which was designed by Johnsson [9]. The work was based on Turner's combinator approach, but instead of using a fixed set of combinators, each user defined function is used as a "combinator", actually a rewrite rule. Functions are compiled into code for the G-machine which, when executed, creates and reduces expression graphs to reduce expressions to their values.

Kieburtz [14] presents an evaluator based upon the G-machine's abstract architecture. He points out that control in a programmed graph reduction is specified by a sequence of instructions. These instructions are statically derived by the compilation of the applicative expression. This contrasts with combinator reduction, where control is derived dynamically from the combinator expression.

It is suggested that a programmed reduction system allows the use of current technology in that current computer architectures can execute the machine code. However, as we have seen, functional languages are not supported very efficiently on von Neumann architectures. The maintenance of and access to the environment appear to be the primary source of overhead when evaluating functional language programs with conventional
evaluators. The Kieburstz evaluator alters the importance of overhead of the environment by eliminating non-local variables, creating suspensions rather than closures, saving state in hardware, utilizing a separate processor to handle memory management, and the use of the register space as stacks.

2.3 Lambda Calculus

2.3.1 Functional Languages Compared to Imperative Languages

Traditional imperative languages force the programmer to think of the flow of control through a program. This is brought on, in part, by thinking of variables as storage locations. Program statements are history sensitive, forcing the value of a variable to be dependent on the order of computation. This demands that the programmer keep track of the interaction different sections of code have on the same variables. This presence of "side effects" reduces the capability of parallelism in execution.

Functional languages avoid some of the problems associated with the conventional languages. There can not be any side effects because the model does not include assignment statements. The programmer, therefore, does not need to be concerned with the flow of control through a program, i.e., the programs are not history sensitive. The value of an expression depends only on its context. As a result, an expression can be evaluated at any time and replaced by its equivalent value.

2.3.2 Functional Languages and the Lambda Calculus

Functional languages are based on a mathematical model known as the lambda calculus. It provides a formal method to study functions and function application. We will use the lambda calculus model to define our functional language.
2.3.3 Applicative Expressions

Functional languages are based on applying a function to its arguments. An applicative expression (AE) is either

1. an identifier,

2. or a λ-expression, which has a bound variable part that is either an identifier or an identifier list, and a λ-body, which is an AE,

3. or a combination, consisting of an operator (rator), and an operand (rand). Both the rator and the rand are AEs.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td>identifier</td>
</tr>
<tr>
<td>(\lambda a\ .\ a + 1)</td>
<td>λ-expression</td>
</tr>
<tr>
<td>(\lambda a\ .\ \lambda b\ .\ a / (2 \times b + 3))</td>
<td>λ-expression</td>
</tr>
<tr>
<td>(\lambda a\ .\ a + 1)(4))</td>
<td>combination</td>
</tr>
</tbody>
</table>

In the combination above, \(\lambda a\ .\ a + 1\) is the rator, or function. It is an AE, with \(\lambda a\) representing the bound variable part, and \(a + 1\) representing the λ-body. The rand of the combination is the \((4)\). Function application occurs by substituting the 4 for every occurrence of \(a\) in the λ-body.

2.3.4 Useful Properties of the Lambda Calculus

There are certain features of the lambda calculus that make the model very useful. One such property, referential transparency, is due to the absence of assignment statements. When an expression is evaluated, it can be replaced by its equivalent value at any time. Unlike updating a variable, there are no possible side effects from replacing the
expression with an equivalent form. This equivalence provides an opportunity to optimize. Once the evaluation is completed for a particular expression, the resulting value can be used later without re-evaluating the original expression. This could result in a savings of time and resources during program execution.

Another feature of the lambda calculus is called the Church-Rosser property [4]. It allows expressions in the lambda calculus to be evaluated in any order. If different orders of evaluation all terminate, we are guaranteed they each yield the same result. This means we can evaluate the rators and the rands in any order. In fact, we may even evaluate the rators and the rands in parallel and apply the results of the rators to the resulting rands.

The following example illustrates the ability to evaluate an expression in different orders. Part (a) of the example uses an outside/in evaluation order, while part (b) uses an inside/out approach.

(a)

\[
[\lambda x . ([\lambda z . ([\lambda y . z \times y ](x + 2))(3))](2)] \\
= [\lambda z . ([\lambda y . z \times y ](2 + 2))(3)] \\
= [\lambda z . ([\lambda y . z \times y ](4))(3)] \\
= [\lambda y . 3 \times y ](4) \\
= 3 \times 4 \\
= 12
\]

(b)

\[
[\lambda x . ([\lambda z . ([\lambda y . z \times y ](x + 2))(3))](2)] \\
= [\lambda x . ([\lambda z . z \times (x + 2)](3))(2)] \\
= [\lambda x . 3 \times (x + 2)](2) \\
= 3 \times (2 + 2) \\
= 3 \times 4 \\
= 12
\]

This next example illustrates the importance of convergence for two different evaluation orders to produce equivalent results. Given the AE:

\[
[\lambda x . 2][\lambda y . y y ](\lambda y . y y )
\]
First, applying the rator to the rand, we evaluate the expression to 2. Now, suppose we choose to evaluate the rand before making the function application. We get:

$$\lambda x \cdot 2((\lambda y \cdot y y)(\lambda y \cdot y y))$$

The rand evaluates to itself! It is easy to see that if we choose to always evaluate the rand in this example, then the evaluation will never converge.

The following example is designed to allow parallel evaluation of the rator and the rand. First, a sequential evaluation is demonstrated in part (a).

(a)

$$[\[x \cdot k \cdot x + 1\](1) \ldots (^2 + 1)] = [\[x \cdot k \cdot x + 1\](1)(^2 + 1)] = [\[y \cdot 1 + 1\](3)] = [y \cdot 2](3) = 2$$

This particular expression could have the rator and the rand evaluated in parallel. One machine could evaluate the rator and then wait for the evaluation of the rand to complete (via a second machine) and finish the expression evaluation by applying the resulting rator to the final form of the rand. Part (b) demonstrates this optimization:

(b)

Machine 1

$$\lambda x \cdot \lambda y \cdot x + 1)(1) = \lambda y \cdot 1 + 1 = 2 + 1$$

Machine 2

$$((\lambda z \cdot z + 1)(2)) = \lambda y \cdot 2$$

To finish, Machine 1 gets the result from Machine 2 and applies the rator to the rand:

$$= [\lambda y \cdot 2](3) = 2$$
2.3.5 Correspondence Between a Functional Language and the Lambda Calculus

A simple functional language consisting of let expressions illustrates a correspondence between functional languages and the lambda calculus. A let expression creates a local environment by binding the variables. The general form of the functional language is:

\[
\text{let } f (x_1, x_2, \ldots, x_n) = E \\
in \\
M
\]

This corresponds to the \(\lambda\)-expression:

\[
[\lambda f . M][\lambda(x_1, x_2, \ldots, x_n) . E)
\]

A simple example illustrates this mapping:

\[
\text{let } f (x) = x + 1 \\
in \\
f (2)
\]

corresponds to

\[
[\lambda f . f (2)][\lambda x . x + 1)
= [\lambda x . x + 1](2)
= 2 + 1
= 3
\]

The next example illustrates the static scoping used in the lambda calculus.

\[
\text{let } a = 1 \\
in \\
\text{let } f (x) = a + 2 \\
in \\
\text{let } a = 5 \\
in \\
f (17)
\]

corresponds to
\[
\lambda a \cdot \lambda f \cdot ((\lambda a \cdot f(17))(5))(\lambda x \cdot a + 2)(1)
\]
\[
= \lambda a \cdot \lambda f \cdot f(17)(\lambda x \cdot a + 2)(1)
\]
\[
= \lambda a \cdot (\lambda x \cdot a + 2)(17)(1)
\]
\[
= \lambda a \cdot a + 2(1)
\]
\[
= 1 + 2
\]
\[
= 3
\]

The environment for the function \( f(x) \) includes the binding of \( a \) to 1. Therefore, when the function is called with an argument of 17, the result is 3. If dynamic scoping had been used the result would have been 5. The next example illustrates the need to rename variables in order to avoid confusion regarding the scope of the bound variables.

\[
\text{let } x = 2 \\
\text{ in } \\
\text{let } x = 3 \\
\text{ in } \\
\text{let } y = x + 2 \\
\text{ in } \\
\text{let } y = x \\
\text{ corresponds to }
\]

\[
[\lambda x . ((\lambda x . (\lambda y . x \times y)(x + 2))(3))](2)
\]

Upon renaming, we get the following:

\[
[\lambda x . ((\lambda z . ((\lambda y . z \times y)(x + 2))(3))](2)
\]
\[
= [\lambda z . ((\lambda y . z \times y)(2 + 2))(3)
\]
\[
= [\lambda z . ((\lambda y . z \times y)(4))(3)
\]
\[
= [\lambda y . 3 \times y](4)
\]
\[
= 3 \times 4
\]
\[
= 12
\]

### 2.4 Combinators

Another representation of functional languages is the combinator model. This model allows us to eliminate the use of variables. The role of these eliminated variables can be expressed by primitive operators, called combinators, together with a mechanism for applying a function to its argument.
2.4.1 Useful Properties of the Combinator Model

Two properties of the combinator model are of interest. First, the evaluation permits non-strict functions to be implemented at no extra cost. This means that not all arguments to a function must be evaluated, i.e., an operand will only be evaluated if and when it is needed. The lambda calculus model does not support non-strict functions unless some modifications are made (at an extra cost). Second, user defined functions are inexpensive to use (after the initial overhead cost of the first call).

2.4.2 Correspondence of the Lambda Calculus and Combinators

We are interested in transforming $\lambda$-expressions into combinator expressions. (We already have seen how to translate a high level functional language into $\lambda$-expressions.) This step will allow us to take a high level language and "compile" it into a form that can be executed by a simple combinator machine. The instruction set of this machine consists of the combinator rules. As we will see, the use of combinators will eliminate the need for bound variables and the environment model to represent them.

Turner discusses a method for removing variables from programs which is based on three initial combinators:

\begin{align*}
S & \colon f \, g \, x = f \, (g \, x) \\
K & \colon x \, y = x \\
I & \colon x = x
\end{align*}

A special abstraction operation denoted by $[x]E$ is used to remove all occurrences of $x$ in $E$, where $x$ is a variable and $E$ is an expression. The abstraction operation is defined for combinators $S$, $K$, and $I$ as follows:

\begin{align*}
[x](E_1, E_2) & \rightarrow S ([x]E_1), ([x]E_2) \\
[x]y & \rightarrow K y \quad x \neq y \\
[x]x & \rightarrow I
\end{align*}
where $E_1$ and $E_2$ are expressions and where $y$ is a constant or variable (other than $x$).

The resulting code from abstracting the variables can be very large. There are some optimization rules:

\[
\begin{align*}
S (K E_j)(K E_j) &\rightarrow K (E_j E_j) \\
S (K E_j) I &\rightarrow E_j \\
S (K E_j) E_j &\rightarrow B E_j E_j \text{ if no earlier rule applies} \\
S E_j (K E_j) &\rightarrow C E_j E_j \text{ if no earlier rule applies}
\end{align*}
\]

These optimization rules account for special cases where the variable being abstracted is not present in all subexpressions. $B$ and $C$ are combinators to handle these special situations.

An example conversion from a $\lambda$-expression to combinator expression illustrates the elimination of the variables via the above rules. The following example was mapped from a simple functional language into a $\lambda$-expression in section 2.3.5.

\[
\lambda f.(f(2))(\lambda x.x+1)
\]

1. Conversion (abstract both $f$ and $x$)

\[
\begin{align*}
&\left[ f (f(2))(\lambda x.x+1) \right] \\
&= (S (f f)(f 2))(S (\lambda x.(x+1))(\lambda x.1)) \\
&= (S (1 I)(K 2))(S (S (\lambda x.(x+1))(\lambda x)(\lambda x.1)) (K 1)) \\
&= (S (1 I)(K 2))(S (S (K K)(I 1))(K 1))
\end{align*}
\]

2. Optimization

\[
\begin{align*}
&= C I 2(S (K 2)(I 1)) \\
&= C I 2(C \text{ plus } 1)
\end{align*}
\]

Notice that the variables $f$ and $x$ have been eliminated from the final expression.
2.4.3 The S-K Reduction Machine

The S-K Reduction Machine evaluates the combinator expressions resulting from the variable abstraction. The evaluation mechanism is through transformation of the combinator expression. Reduction rules are applied to each stage of the transformation until a value is obtained. Some of the reduction rules are:

\[ S \, f \, g \, x \rightarrow f \, (g \, x) \]
\[ K \, x \, y \rightarrow x \]
\[ C \, f \, g \, x \rightarrow (f \, x) \, g \]
\[ B \, f \, g \, x \rightarrow f \, (g \, x) \]
\[ I \, x \rightarrow x \]

and also the primitives, such as:

\[ \text{plus} \, x \, y \rightarrow x + y \]

etc.

To continue with the example from the above section:

3. Reduction

\[ = I \, (\, C \, \text{plus} \, 1) \, 2 \]
\[ = C \, \text{plus} \, 1 \, \, 2 \]
\[ = \text{plus} \, 2 \, \, \, 1 \]
\[ = 3 \]

The S-K reduction machine transforms the combinator expression with each reduction step. This is unlike a conventional fixed program machine that executes the code, usually without modification. The transformations that occur replace expressions by mathematically equivalent ones, e.g., plus 1 2 is replaced by 3 so that the actual evaluation of plus 1 2 only occurs once.

The order of the reductions is defined as normal order. At each step, the leftmost reduction is performed. This method is not only simple to use, but is guaranteed to terminate (if possible). Normal order reduction is what we previously called the
outside/in evaluation. The other order we looked at, inside/out, is called applicative order reduction. This involves performing the innermost reductions first. Applicative order reduction is not guaranteed to converge. Also, applicative order reduction does not support non-strict functions. Each subexpression must be evaluated, which can cause problems.

The normal order reduction method supports non-strict functions.
3. A METHODOLOGY TO GENERATE COMPUTER ARCHITECTURES

3.1 Abdali’s Abstraction Algorithm

In an attempt to improve upon Turner’s combinator sequences, Abdali presents an algorithm to carry out a new abstraction method. His method results in combinator code consisting of a different set of combinators. It performs a one step abstraction on multiple variables and yields a single combinator for the group of variables. In comparison, Turner’s method requires that the abstractions be nested and a combinator sequence be generated for each variable being abstracted.

To accommodate multiple variable abstractions, Abdali introduces the notion of a family of combinators. Each combinator family has an infinite number of members, but each member behaves in a manner dictated by the family. Consider, for example, the $K$ combinator that Turner uses. Its behavior can be characterized by the reduction rule $K \ x \ y \rightarrow x$. Abdali makes use of a family of $K_n$ combinators, where Turner’s $K$ combinator is actually a member of the set, in particular $K_1$. The action of the $K_n$ combinator family is given by the reduction rule:

$$K_n \ a \ b_1 \ldots \ b_n \rightarrow a \quad n \geq 1$$

It is easy to see that Turner’s $K$ follows the family’s reduction rule, for $n = 1$.

The other combinator families employed in Abdali’s scheme contain members whose actions we are already familiar with:

$$I \ x \rightarrow x$$

$$B \ f \ x \ y \rightarrow f (x \ y)$$

The more general family combinators have the following reduction properties:

$$I_n^m \ a_1 \ldots a_n \rightarrow a_m \quad n \geq m \geq 1$$

$$B_n^m \ a \ b_1 \ldots b_m \ c_1 \ldots c_n \rightarrow a \ (b_1 \ c_1 \ldots c_n) \ldots (b_m \ c_1 \ldots c_n) \quad m \cdot n \geq 1$$
By inspection, we can see that the combinators we are familiar with are indeed members of the above families. Suppose \( m = 1 \) and \( n = 1 \):

\[
I_1^1 \ x \rightarrow x
\]

\[
B_1^1 \ f \ x \ y \rightarrow f \ (x \ y)
\]

A working description of Abdali's abstraction algorithm will be given. For a more formal treatment, see the original work [1]. An expression of the form \([x_1 \cdots x_n]e\) indicates that the variables \(x_1 \cdots x_n\) are being abstracted from the expression \(e\). Given such an expression, the following rules are applied, in order, to perform the abstraction.

1. If \(x_i\) is not contained in \(e\) for all \(i\), then

\[
[x_1 \cdots x_n]e \rightarrow K_n \ e \quad 1 \leq i \leq n
\]

Examples:

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

\[
[x,y] 7 \rightarrow K_2 \ 7
\]

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

2. If \(e\) matches \(x_1 \cdots x_n\), then

\[
[x_1 \cdots x_n]e \rightarrow I
\]

Examples:

\[
[x] \ x \rightarrow I
\]

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

3. If \(e\) matches an \(x_i\) for some \(i\), then

\[
[x_1 \cdots x_n]e \rightarrow I_i^i \quad 1 \leq i \leq n
\]

Examples:

\[
[x,y] \ x \rightarrow I_1^i
\]

\[
[x,y,x] \ y \rightarrow I_2^i
\]
4. If $e$ is of the form $g \cdot x_1 \cdots x_n$ and $x_i$ is not contained in $g$ for all $i$, then

$$[x_1 \cdots x_n] \rightarrow g \quad 1 \leq i \leq n$$

Examples:

$$[x] + x \rightarrow +$$
$$[x,y] + x \cdot y \rightarrow +$$

5. If $e$ is of the form $g \cdot x_m \cdots x_n$ and $x_i$ is not contained in $g$ for $i \geq m$, then

$$[x_1 \cdots x_n] \rightarrow [x_1 \cdots x_{m-1}] g$$

Examples:

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

6. If $e$ is of the form $x_i \cdot f_2 \cdots f_m$ for some $i$, then

$$[x_1 \cdots x_n] \rightarrow B_n^{m-1} I I_1^i \cdot ([x_1 \cdots x_n] f_2) \cdots ([x_1 \cdots x_n] f_m)$$

Examples:

$$[x] x a b c \rightarrow B_{1}^4 I I_1^1 ([x] a) ([x] b) ([x] c)$$
$$[x,y] x a b c \rightarrow B_{2}^4 I I_2^2 ([x,y] a) ([x,y] b) ([x,y] c)$$
$$[x,y] x a b \rightarrow B_{3}^2 I I_3^2 ([x,y] a) ([x,y] b)$$
$$[f] / 2 \rightarrow B_{4}^2 I I_4^1 ([f] 2)$$

7. If $e$ is of the form $f_1 \cdot f_2 \cdots f_m$ where $f_1$ is the longest initial component not containing an $x_i$ for all $i$, then

$$[x_1 \cdots x_n] \rightarrow B_n^{m-1} f_1 ([x_1 \cdots x_n] f_2) \cdots ([x_1 \cdots x_n] f_m)$$

Examples:

$$[x] a b c \rightarrow B_{1}^2 a ([x] b c) ([x] c)$$
$$[x,y] a b (c y) (d x) \rightarrow B_{2}^3 a b ([x,y] c y) ([x,y] d x)$$
$$[x] + x 1 \rightarrow B_{1}^2 + ([x] x) ([x] 1)$$
3.1.1 Demonstration of Abdali's Abstraction Method

A few examples of converting simple functional programs into combinator code will illustrate how Abdali's abstraction method works. Later in this chapter we will compare combinator sequences obtained by Abdali's method to those obtained when Turner's original abstraction algorithm is followed.

Example 1 (Function Definition)

\[
\text{let } f(x) = x + 1 \\
\text{in } f(2)
\]

Lambda calculus notation: \[\lambda f . f \ 2 \ ] (\lambda x . + x \ 1)

Abdali's abstraction method:

\[
[[f \ f \ 2 \ ] ([x] + x \ 1)]
\]

Rule 6 \[B_1^2 \ \ I \ ] \[([f \ ] \ 2) ([x] + x \ 1)\]

Rule 1 \[B_1^2 \ \ I \ ] \[([K_1 \ 2] ([x] + x \ 1)\]

Rule 7 \[B_1^2 \ \ I \ ] \[([K_1 \ 2] \ B_1^2 + ([x] \ x) ([x] \ 1))\]

Rule 2 \[B_1^2 \ \ I \ ] \[([K_1 \ 2] \ B_1^2 + I ([x] \ 1))\]

Rule 1 \[B_1^2 \ \ I \ ] \[([K_1 \ 2] \ (B_1^2 + I (K_1 \ 1))\]

Reduction:

\[
B_1^2 \ \ I \ ] \[([K_1 \ 2] \ (B_1^2 + I (K_1 \ 1))\]
\[
\rightarrow I \ [[B_1^2 + I (K_1 \ 1))] \ K_1 \ 2 (B_1^2 + I (K_1 \ 1))\]
\[
\rightarrow I \ ([B_1^2 + I (K_1 \ 1))] \ K_1 \ 2 (B_1^2 + I (K_1 \ 1))\]
\[
\rightarrow B_1^2 + I (K_1 \ 1) (K_1 \ 2 (B_1^2 + I (K_1 \ 1)))\]
\[
\rightarrow + (I (K_1 \ 2 (B_1^2 + I (K_1 \ 1)))) (K_1 \ 1 (K_1 \ 2 (B_1^2 + I (K_1 \ 1))))\]
\[
\rightarrow + (K_1 \ 2 (B_1^2 + I (K_1 \ 1))) \ 1\]
\[
\rightarrow 2 \ 1\]
\[
\rightarrow 3
\]
Example 2 (Multiple Parameters)

let \( f \ (a \ . \ b) = a + b \)

in

\( f \ (3,4) \)

Lambda calculus notation: \([\lambda f \ . \ f \ (3 \ 4)] (\lambda (a \ . \ b) \ . \ a + b)\)

Abdali's abstraction method:

\((([f] \ f \ 3 \ 4) ([a \ . \ b] + a \ b)\)

Rule 6 \( B^1_1 I^1 I^1 \ ([f] \ 3) ([f] \ 4) ([a \ . \ b] + a \ b)\)

Rule 1 \( B^1_1 I^1 I^1 \ (K_1 \ 3) ([f] \ 4) ([a \ . \ b] + a \ b)\)

Rule 1 \( B^1_1 I^1 I^1 \ (K_1 \ 3) (K_1 \ 4) ([a \ . \ b] + a \ b)\)

Rule 4 \( B^1_1 I^1 I^1 \ (K_1 \ 3) (K_1 \ 4) +\)

Reduction:

\( B^1_1 I^1 I^1 \ (K_1 \ 3) (K_1 \ 4) +\)

\( \rightarrow I \ (I^1 +) (K_1 \ 3 +) (K_1 \ 4 +)\)

\( \rightarrow I^1 + (K_1 \ 3 +) (K_1 \ 4 +)\)

\( \rightarrow + (K_1 \ 3 +) (K_1 \ 4 +)\)

\( \rightarrow + 3 (K_1 \ 4 +)\)

\( \rightarrow + 3 4\)

\( \rightarrow 7\)

Example 3 (Nesting)

let \( a = 1 \)

in

let \( b = 2 \)

in

\( a + b\)

Lambda calculus notation: \([\lambda a \ . \ [\lambda b \ . \ a + b] \ 2] \ 1\)

Abdali's abstraction method:

\(([[a] \ ([b] + a \ b) \ 2] \ 1\)

Note the abstraction is performed from the inside of the expression to the outside.
3.1.2 Comparison of Abdali's Method and Turner's Method

The primary reason we became interested in Abdali's work was that we were looking for a way to use simultaneous definitions in the lambda calculus and still have the capability to convert programs into combinator sequences. Consider the following program:

\[
\text{let } a = 1 \text{ and } b = 2 \\
\text{in} \\
a + b
\]

Turner's method would force us to curry the lambda calculus notation for this program:

\[
[\lambda a . [\lambda b . (+ a) b] 2] 1
\]

The above notation is identical to what we would arrive at for

\[
\text{let } a = 1 \\
\text{in} \\
\text{let } b = 2 \\
\text{in} \\
a + b
\]

Thus, the necessity of currying forces the nesting of the variables where nesting is not
really desired.

What we want is for the variables \(a\) and \(b\) to be declared in the same block, and the abstractions to clearly depict the simultaneous nature of the variable declarations. Suppose we represent the above program as

\[
[\lambda(a . b) . + \ a \ b]\ (1.2)
\]

where the \((a . b)\) represents the simultaneous definition of the variables. It is now an easy job to translate the lambda calculus notation into a combinator sequence using Abdali's method.

\[
([a . b] + a . b)\ (1.2)
\]

Rule 4 + (1.2)

We won't always see such a simple outcome, but it is clear that Abdali's method will handle simultaneous definitions as well as multiparameter functions.

To further demonstrate this multiparameter capability. Example 2 above shows that

\[
\text{let } f \ (a . b) = a + b
\]

\[
in\ f \ (3.4)
\]

translates to \([\lambda f . f \ 3 \ 4] (\lambda(a . b) . + \ a \ b)\). This lambda calculus equation can easily be handled by Abdali's abstraction method. In order to apply Turner's abstraction rules, the equation must be curried to \([\lambda f . (f \ 3) \ 4] (\lambda a . (\lambda b . (+ a \ b)))\). This currying operation forces us to abstract \(a\) and \(b\) in sequence rather than simultaneously, thus making the abstraction process a little longer.

Lambda calculus notation: \([\lambda f . (f \ 3) \ 4] (\lambda a . (\lambda b . (+ a \ b)))\)

Turner's abstraction method:

\[
([f](f \ 3) \ 4) ([a] ([b] (+ a \ b))
\]

\[
(S ([f](f \ 3) ([f] 4)) ([a] (S ([b] + a) ([b] b)))
\]

\[
(S (S ([f] f) ([f] 3)) (K 4)) ([a] (S (S ([b] +) ([b] a)) f))
\]
Upon further study of Abdali versus Turner combinator representations of programs, it became apparent that in certain cases Abdali’s representations were inferior to Turner’s due to the sheer number of combinators generated. Clearly Abdali’s method offers the capability to handle multiple parameters while Turner’s algorithm does not, but in some situations Turner’s code is more desirable. For example, consider Example 3 from above.

```
let a = 1
in
let b = 2
in
  a + b
```

Abdali’s method generates the combinator sequence $B^2_1 + I (K \, 2) \, 1$. Turner’s optimized code for this program is much shorter: $C \, + \, 2 \, 1$. 
3.2 Introduction of New Combinators

The logical step was to somehow combine the features of both methods. We wanted reasonable combinator sequences to be generated, but did not want to lose the capability to handle multiple parameters simultaneously.

Turner's code is an optimized version, so we attempted to model our new abstraction scheme on Turner's set of combinators. At the same time, we decided to keep the notion of families of combinators that Abdali uses. Turner's rules for abstraction are repeated again below.

\[
[x](E_1 E_2) \to S ([x] E_1) ([x] E_2)
\]
\[
[x] x \to I
\]
\[
[x] y \to K y
\]

Abdali includes the \( I \) and \( K \) in his translation rules as follows:

For \([x_1 \cdots x_n] e\)

1. If \( x_i \) is not contained in \( e \) for all \( i \), then

\[
[x_1 \cdots x_n] e \to K_n e \quad 1 \leq i \leq n
\]

2. If \( e \) matches \( x_1 \cdots x_n \), then

\[
[x_1 \cdots x_n] e \to I
\]

3. If \( e \) matches an \( X_i \) for some \( i \), then

\[
[x_1 \cdots x_n] e \to I'_n \quad 1 \leq i \leq n
\]

We introduce an Abdali-like translation rule for \( S''_n \):

If \( e \) is an application \( F A \), then

\[
[x_1 \cdots x_n] e \to S'_1 ([x_1 \cdots x_n] F) ([x_1 \cdots x_n] A)
\]

Extending this rule for the more general case, we get the following:
If \( e \) is an application grouping \( f_1 f_2 \cdots f_m A \), then

\[
[x_1 \cdots x_n] e \rightarrow S_n^m ([x_1 \cdots x_n] f_1) ([x_1 \cdots x_n] f_2) \cdots ([x_1 \cdots x_n] f_m) ([x_1 \cdots x_n] A)
\]

The \( S_n^m \) combinator is defined by

\[
S_n^m x y_1 \cdots y_m z_1 \cdots z_n \rightarrow (x z_1 \cdots z_n)(y_1 z_1 \cdots z_n) \cdots (y_m z_1 \cdots z_n)
\]

It is easy to see that the \( S \) combinator Turner uses is a member of the \( S_n^m \) family.

\[
S f g x = S_1^1 f g x = f x (g x)
\]

We now have three families of combinators: \( S_n^m, K_n \), and \( I_n^m \). We consider \( I \) as a special case of the family \( I_n^m \). Turner used members of these families. While they are sufficient to remove all variables from programs, long-winded combinator code results. To improve the code, extra combinators \( B \) and \( C \) were introduced. We will also include the families of \( B_n^m \) and \( C_n^m \) in our new set of combinators.

Abdali already introduced the \( B_n^m \) family of combinators in his work.

\[
B_n^m a b_1 \cdots b_m c_1 \cdots c_n \rightarrow a (b_1 c_1 \cdots c_n) \cdots (b_m c_1 \cdots c_n)
\]

The \( C_n^m \) combinator family is similarly defined.

\[
C_n^m f_1 \cdots f_m g x_1 \cdots x_n \rightarrow (f_1 x_1 \cdots x_n) \cdots (f_m x_1 \cdots x_n) g
\]

The optimizations for these combinators, patterned after Turner’s results, follow:

Opt 1

\[
S_n^m (K_n E_0) (K_n E_1) \cdots (K_n E_m) \rightarrow K_n (E_0 E_1 \cdots E_m)
\]

Opt 2

\[
S_n^m (K_n E_0) I_n^1 I_n^2 \cdots I_n^n \rightarrow E_0
\]

Opt 3

\[
S_n^m (K_n E_0) E_1 \cdots E_m \rightarrow B_n^m E_0 E_1 \cdots E_m
\]

Opt 4

\[
S_n^m E_0 E_1 \cdots E_{m-1} (K_n E_m) \rightarrow C_n^m E_0 E_1 \cdots E_m
\]
We can show that the left and right hand sides of these equations are always equal by applying both sides to arbitrary \( x_1 \cdots x_n \) and simplifying.

**Opt 1**

\[
S_n^m (K_n E_0) (K_n E_1) \cdots (K_n E_m) x_1 \cdots x_n
\]

\[
\rightarrow (K_n E_0 x_1 \cdots x_n) (K_n E_1 x_1 \cdots x_n) \cdots (K_n E_m x_1 \cdots x_n)
\]

\[
\rightarrow E_0 E_1 \cdots E_m
\]

and

\[
K_n (E_0 E_1 \cdots E_m) x_1 \cdots x_n
\]

\[
\rightarrow E_0 E_1 \cdots E_m
\]

therefore

\[
S_n^m (K_n E_0) (K_n E_1) \cdots (K_n E_m) \rightarrow K_n (E_0 E_1 \cdots E_m)
\]

**Opt 2**

\[
S_n^m (K_n E_0) I_n^1 I_n^2 \cdots I_n^n x_1 \cdots x_n
\]

\[
\rightarrow (K_n E_0 x_1 \cdots x_n) (I_n^1 x_1 \cdots x_n) (I_n^2 x_1 \cdots x_n) \cdots (I_n^n x_1 \cdots x_n)
\]

\[
\rightarrow E_0 x_1 x_2 \cdots x_n
\]

and

\[
E_0 x_1 x_2 \cdots x_n
\]

therefore

\[
S_n^m (K_n E_0) I_n^1 I_n^2 \cdots I_n^n x_1 \cdots x_n \rightarrow E_0
\]

**Opt 3**

\[
S_n^m (K_n E_0) E_1 \cdots E_m x_1 \cdots x_n
\]

\[
\rightarrow (K_n E_0 x_1 \cdots x_n) (E_1 x_1 \cdots x_n) \cdots (E_m x_1 \cdots x_n)
\]

\[
\rightarrow E_0 (E_1 x_1 \cdots x_n) \cdots (E_m x_1 \cdots x_n)
\]

and

\[
B_n^m E_0 E_1 \cdots E_m x_1 \cdots x_n
\]
\[ \rightarrow E_0 (E_1 x_1 \cdots x_n) \cdots (E_m x_1 \cdots x_n) \]

therefore

\[ S^n_m (K_n E_0) E_1 \cdots E_m \rightarrow B^n_m E_0 E_1 \cdots E_m x_1 \cdots x_n \]

Opt 4

\[ S^n_m E_0 E_1 \cdots E_{m-1} (K_n E_m) x_1 \cdots x_n \]

\[ \rightarrow (E_0 x_1 \cdots x_n) (E_1 x_1 \cdots x_n) \cdots (E_{m-1} x_1 \cdots x_n) (K_n E_m x_1 \cdots x_n) \]

\[ \rightarrow (E_0 x_1 \cdots x_n) (E_1 x_1 \cdots x_n) \cdots (E_{m-1} x_1 \cdots x_n) E_m \]

and

\[ C^n_m E_0 E_1 \cdots E_m x_1 \cdots x_n \]

\[ \rightarrow (E_0 x_1 \cdots x_n) (E_1 x_1 \cdots x_n) \cdots (E_{m-1} x_1 \cdots x_n) E_m \]

therefore

\[ S^n_m E_0 E_1 \cdots E_{m-1} (K_n E_m) \rightarrow C^n_m E_0 E_1 \cdots E_m \]

### 3.2.1 Revisiting Examples Using New Abstraction Rules

It will be useful to study some examples to see if the proposed set of abstraction rules produce desirable code sequences. We will revisit some earlier examples.

**Example 1**

\[
\text{let } f \ (x) = x + 1 \\
\text{in} \\
\quad f \ (2)
\]

Lambda calculus notation: \( [\lambda f . f \ 2] (\lambda x . x + 1) \)

New abstraction method:

\[
([f \ f \ 2]) ([x \ x \ + \ 1])
\]

\( S^1_I ([f \ f \ 2]) (S^2_I ([x \ x \ + \ 1])) \)

\( S^1_I I \ (K_1 \ 2) (S^2_I (K_1 \ + \ I \ K_1 \ 1)) \)

optimizing once yields
\[ S_1 I (K_1 2) (B_1^2 + I (K_1 1)) \]
and optimizing again yields
\[ C_1 I 2(B_1^2 + I (K_1 1)) \]
Reduction:
\[
\begin{align*}
C_1 I 2(B_1^2 + I (K_1 1)) \\
\to (I (B_1^2 + I (K_1 1))) 2 \\
\to B_1^2 + I (K_1 1) 2 \\
\to + (I 2) (K_1 1 2) \\
\to + 2 (K_1 1 2) \\
\to + 2 1 \\
\to 3
\end{align*}
\]
One way of measuring the desirability of the code produced is to count the number
of combinators that result from each method.

New abstraction method: \[ C_1 I 2(B_1^2 + I (K_1 1)) \]
Abdali's method: \[ B_1^2 I I_1 (K_1 2) (B_1^2 + I (K_1 1)) \]
Turner's method: \[ C I 2(C + 1) \]
In this example, we see that our new method produced code with a number of combinators
between the number resulting from Turner's method and from Abdali's method. Since we
have added the ability to handle multiple parameters and simultaneous definitions to
Turner's original method, we should expect to pay the price by producing longer code.
Similarly, since we have introduced optimizations to Abdali's technique, we would expect
to get code that is the same length (if the Abdali code is optimal) or shorter. The cost to
have the shorter code is a longer compilation phase, but the method gives us (potentially)
a shorter execution phase.
Example 2  

\[
\text{let } f \ (a \ b) = a + b \ \\
\text{in } f \ (3,4)
\]

Lambda calculus notation: \[
[\lambda f \ . \ f \ (3 \ 4)] \ (\lambda(a \ b) . \ + a b)
\]

New abstraction method:

\[
([f ] \ f \ 3 \ 4) \ ([a \ b] (+ a b))
\]

\[
S ^ 2 \ ([f ] \ f \ ) \ ([f ] \ 3) \ ([f ] \ 4) \ (S ^ 2 \ ([a \ b] +) \ ([a \ b] a) \ ([a \ b] b))
\]

optimizing

\[
S ^ 2 \ I \ (K_1 \ 3) \ (K_1 \ 4) \ (S ^ 2 \ (K_2 +) \ I ^ 2 \ I ^ 2)
\]

optimizing

\[
C ^ 2 \ I \ (K_1 \ 3) \ 4 +
\]

Abdali’s method: \[
B ^ 1 \ I \ I ^ 1 \ (K_1 \ 3) \ (K_1 \ 4) +
\]

We have previously discussed the necessity of currying in order for Turner’s original method to handle the above example. Turner’s abstraction process requires more work than our new technique.

Turner’s method: \[
C \ (C \ I \ 3) \ 4 +
\]

Note in this example that our new technique produces code that is just as efficient as Turner’s, yet is produced in fewer abstraction steps.

Reduction:

\[
C ^ 2 \ I \ (K_1 \ 3) \ 4 +
\]

\[
\rightarrow (I +) \ (K_1 \ 3 +) \ 4
\]

\[
\rightarrow + \ (K_1 \ 3 +) \ 4
\]

\[
\rightarrow + \ 3 \ 4
\]

\[
\rightarrow 7
\]
Example 3

\[
\text{let } a = 1 \\
\text{in} \\
\text{let } b = 2 \\
\text{in} \\
a + b
\]

Lambda calculus notation: \[
(\lambda a \cdot (\lambda b \cdot + a b)) 2 1
\]

New abstraction method:

\[\frac{\lambda a}{[(a) (\lambda b . (\lambda c . + b c)) (\lambda x . x)]}{(1) (2)}\]

optimizing

\[\frac{\lambda a}{[(a) (\lambda b . (\lambda c . + b c))]{(1) (2)}}\]

optimizing

\[\frac{\lambda a}{[(a) (\lambda b . (\lambda c . + b c))]{(1) (2)}}\]

optimizing

\[\frac{\lambda a}{[(a) (\lambda b . (\lambda c . + b c))]{(1) (2)}}\]

optimizing

\[\frac{\lambda a}{[(a) (\lambda b . (\lambda c . + b c))]{(1) (2)}}\]

Abdali’s method: \[B 2 1\]

Turner’s method: \[C + 2 1\]

Our method obviously has some faults also! The Abdali generated code is shorter than the code produced by the new abstraction method. One reason for this unexpected result is that the new method is designed to handle simultaneous definitions well, rather than the
nested declarations that this example contains.

3.3 Optimization of the New Abstraction Method

After observing several cases where Abdali’s code was better in some sense than our new code, we decided that our method could be improved upon. The optimization rules for the new method were too restrictive in certain cases. Abdali’s rules could handle some abstractions more efficiently than our current set of rules could. We studied examples to determine some of the inefficiencies of our method. We found that we could improve upon the optimization rules that handled the special cases of recognizing when a variable is being abstracted from one or more subexpressions.

To accommodate this situation, we introduce the combinator families of $X_{n}^{m,k}$ and $Y_{n}^{m,k}$.

\[
X_{n}^{m,k} (y_0 \cdots y_m x_1 \cdots x_n) \rightarrow (y_0 \cdots y_{k-1} (y_k x_1 \cdots x_n) \cdots (y_m x_1 \cdots x_n))
\]

\[
Y_{n}^{m,k} (y_0 \cdots y_m x_1 \cdots x_n) \rightarrow (y_0 x_1 \cdots x_n) \cdots (y_{k-1} x_1 \cdots x_n) y_k \cdots y_m
\]

The optimization rules that produce these combinators are as follows:

Opt 5

\[
S_{n}^m (K_n E_0) \cdots (K_n E_{k-1}) E_k \cdots E_m \rightarrow X_{n}^{m,k} E_0 \cdots E_m
\]

Opt 6

\[
S_{n}^m E_0 \cdots E_{k-1} (K_n E_k) \cdots (K_n E_m) \rightarrow Y_{n}^{m,k} E_0 \cdots E_m
\]

As we did for the first set of optimization rules, we can show that these rules hold by applying both sides of the equations to arbitrary $x_1 \cdots x_n$ and simplifying.

Opt 5

\[
S_{n}^m (K_n E_0) \cdots (K_n E_{k-1}) E_k \cdots E_m x_1 \cdots x_n
\]

\[
\rightarrow (K_n E_0 x_1 \cdots x_n) \cdots (K_n E_{k-1} x_1 \cdots x_n) (E_k x_1 \cdots x_n) \cdots (E_m x_1 \cdots x_n)
\]
→ \( E_0 \cdots E_{k-1} (E_k x_1 \cdots x_n) \cdots (E_m x_1 \cdots x_n) \)

and

\( X_n^{m,k} E_0 \cdots E_m x_1 \cdots x_n \)

→ \( E_0 \cdots E_{k-1} (E_k x_1 \cdots x_n) \cdots (E_m x_1 \cdots x_n) \)

therefore

\[ S_n^m (K_n E_0) \cdots (K_n E_{k-1}) E_k \cdots E_m \rightarrow X_n^{m,k} E_0 \cdots E_m \]

Opt 6

\[ S_n^m E_0 \cdots E_{k-1} (K_n E_k) \cdots (K_n E_m) x_1 \cdots x_n \]

→ \( (E_0 x_1 \cdots x_n) \cdots (E_{k-1} x_1 \cdots x_n) (K_n E_k x_1 \cdots x_n) \cdots (K_n E_m x_1 \cdots x_n) \)

→ \( (E_0 x_1 \cdots x_n) \cdots (E_{k-1} x_1 \cdots x_n) E_k \cdots E_m \)

and

\( Y_n^{m,k} E_0 \cdots E_m x_1 \cdots x_n \)

→ \( (E_0 x_1 \cdots x_n) \cdots (E_{k-1} x_1 \cdots x_n) E_k \cdots E_m \)

therefore

\[ S_n^m E_0 \cdots E_{k-1} (K_n E_k) \cdots (K_n E_m) \rightarrow Y_n^{m,k} E_0 \cdots E_m \]

The new combinator families actually can replace the old combinator optimizations.

Consider, for example, the optimization

\[ S_n^m (K_n E_0) (K_n E_1) \cdots (K_n E_m) \rightarrow K_n (E_0 E_1 \cdots E_m) \]

This can be optimized using the \( X_n^{m,k} \) combinator family.

\[ S_n^m (K_n E_0) (K_n E_1) \cdots (K_n E_m) \rightarrow X_n^{m,m+1} E_1 \cdots E_m \]

Since

\[ S_n^m (K_n E_0) (K_n E_1) \cdots (K_n E_m) x_1 \cdots x_n \]

→ \( (K_n E_0 x_1 \cdots x_n) (K_n E_1 x_1 \cdots x_n) \cdots (K_n E_m x_1 \cdots x_n) \)

→ \( E_0 \cdots E_m \)

and

\( X_n^{m,m+1} E_0 \cdots E_m x_1 \cdots x_n \)
Thus, the optimization rule (Opt 1)

\[ S_n^m (K_n E_0) (K_n E_1) \cdots (K_n E_m) \rightarrow K_n (E_0 E_1 \cdots E_m) \]

can be replaced by the \( X_n^m \) family. This is also true for the rule (Opt 3).

\[ S_n^m (K_n E_0) E_1 \cdots E_m \rightarrow B_n E_0 E_1 \cdots E_m \]

Since

\[ S_n^m (K_n E_0) E_1 \cdots E_m x_1 \cdots x_n \]

\[ \rightarrow (K_n E_0 x_1 \cdots x_n) (E_1 x_1 \cdots x_n) \cdots (E_m x_1 \cdots x_n) \]

\[ \rightarrow E_0 (E_1 x_1 \cdots x_n) \cdots (E_m x_1 \cdots x_n) \]

and

\[ X_n^{m,1} E_0 \cdots E_m x_1 \cdots x_n \]

\[ \rightarrow E_0 (E_1 x_1 \cdots x_n) \cdots (E_m x_1 \cdots x_n) \]

Similarly, the \( Y_n^{m,1} \) combinator optimization can replace optimization rules 1 and 4.

We have chosen to use the \( X_n^{m,1} \) optimization for rule 1. The fourth rule, however, can be replaced since

\[ S_n^m E_0 E_1 \cdots E_{m-1} (K_n E_m) x_1 \cdots x_n \]

\[ \rightarrow (E_0 x_1 \cdots x_n) \cdots (E_{m-1} x_1 \cdots x_n) (K_n E_m x_1 \cdots x_n) \]

\[ \rightarrow (E_0 x_1 \cdots x_n) \cdots (E_{m-1} x_1 \cdots x_n) E_m \]

and

\[ Y_n^{m,1} E_0 \cdots E_{m-1} E_m x_1 \cdots x_n \]

\[ \rightarrow (E_0 x_1 \cdots x_n) \cdots (E_{m-1} x_1 \cdots x_n) E_m \]

Thus, \( S_n^m E_0 E_1 \cdots E_{m-1} (K_n E_m) \rightarrow C_n^m E_0 E_1 \cdots E_m \) can be replaced by the new rule 6.

We will take advantage of these replacement optimizations only to reduce the number of optimization rules required.
3.3.1 Evaluation of the Optimized Abstraction Method

We can assess these new rules by working through the previous examples.

Example 1

\[
\text{let } f(x) = x + 1 \\
\text{in } \\
f(2)
\]

Lambda calculus notation: \([\lambda f . f \ 2] (\lambda x . + x 1)\)

Abstraction:

\[
(f \ f) (\lambda x . + x 1)
\]

\[
S^1 (f \ f) (f \ 2) (S^2 (\lambda x . +) (\lambda x . x) (\lambda x . 1))
\]

\[
S^1 I (K_1 2) (S^2 (K_1 +) I (K_1 1))
\]

optimizing

\[
Y^1 \ I \ 2 (X^2 1 + I (K_1 1))
\]

This combinator sequence contains the same number of combinators as produced by the earlier version of our method.

Reduction:

\[
\rightarrow I (X^2 1 + I (K_1 1)) 2
\]

\[
\rightarrow X^2 I (K_1 1) 2
\]

\[
\rightarrow + (I 2) (K_1 1 2)
\]

\[
\rightarrow + 2 1
\]

\[
\rightarrow 3
\]

Example 2

\[
\text{let } f(a, b) = a + b \\
\text{in } \\
f(3, 4)
\]

Lambda calculus notation: \([\lambda f . f \ (3 \ 4)] (\lambda(a, b). + a b)\)
Abstraction:

\[
(f \ f \ 3 \ 4) \ ((a \ b) \ (+ \ a \ b))
\]
\[
S^2 \ (f \ f) \ (f \ 3) \ (f \ 4) \ (S^2 \ ([a \ b] \ (+) \ ([a \ b] \ a) \ ([a \ b] \ b))
\]
\[
S^2 \ I \ (K_1 \ 3) \ (K_1 \ 4) \ (S^2 \ (K_2) \ I_2) \ I_2
\]

optimizing

\[
Y^2_3 \ I \ 3 \ 4 \ +
\]

In this example, the revised method produces code that is more efficient than any of the three methods we have previously discussed. This is because of the more general optimization that is allowed through the introduction of the \(Y^m_A\) combinator family.

Example 3

\[
\begin{aligned}
\text{let } a &= 1 \\
\text{in} & \\
\text{let } b &= 2 \\
\text{in} & \\
a + b
\end{aligned}
\]

Lambda calculus notation: \([\lambda a \ . \ (\lambda b \ . \ + a \ b)] \ 2 \ 1\]

Abstraction:

\[
([a] \ ([b] \ (+) \ a \ b) \ 2) \ 1
\]
\[
([a] \ (S^2 \ ([b] \ (+) \ ([b] \ a) \ ([b] \ b)) \ 2) \ 1
\]
\[
([a] \ (S^2 \ (K_1) \ (+) \ (K_1 \ a) \ I) \ 2) \ 1
\]

optimizing

\[
([a] \ (X_1^{22} \ (+) \ a \ I) \ 2) \ 1
\]
\[
S^1 \ ([a] \ X_1^{22} \ (+) \ a \ I) \ ([a] \ 2) \ 1
\]
\[
S^1 \ (S^2 \ ([a] \ X_1^{22}) \ ([a] \ (+) \ ([a] \ a) \ ([a] \ I)) \ (K_1 \ 2) \ 1
\]
\[
S^1 \ (S^2 \ (K_1 \ X_1^{22}) \ (K_1 \ (+) \ I \ (K_1 \ I)) \ (K_1 \ 2) \ 1
\]

optimizing
Again, we find that even with the new optimization rules, the resulting combinator sequence is longer than Abdali's and Turner's. The deficiency in the set of optimizations is the inability to handle an optimization of a sequence of the following form:

\( S_1 \) (\( X_1 \) \( X_1^2 \) \( K_1 \) \( I \)) \( 2 \) \( 1 \) \optimizing \( Y_1 \) (\( X_1 \) \( X_1^2 \) \( K_1 \) \( I \)) \( 2 \) \( 1 \)

Such a form is contained in the example above:

\( S_1 \) (\( S_1 \) (\( K_1 \) \( X_1^2 \) \( K_1 \) \( I \)) \( 2 \) \( 1 \)) \( 2 \) \( 1 \)

### 3.4 Another Optimization

We present still another family of combinators designed to handle the deficiency noted above. This new combinator family will be called \( Z_n^{m,k} \), and appears to be quite similar to the \( X_n^{m,k} \) and \( Y_n^{m,k} \) families introduced earlier. This family is defined by the following:

\( Z_n^{m,k} \) \( y_0 \) \( \cdots \) \( y_m \) \( x_1 \) \( \cdots \) \( x_n \) \( \rightarrow \) \( y_0 \) \( \cdots \) \( y_{k-1} \) \( y_k \) \( x_1 \) \( \cdots \) \( x_n \) \( y_{k+1} \) \( \cdots \) \( y_m \)

The optimization rule that creates the \( Z_n^{m,k} \) combinator is as follows:

\optimization\ \( S_n^{m} \) (\( K_n \) \( E_0 \)) \( \cdots \) (\( K_{k-1} \) \( E_k \)) \( K_k \) \( E_k+1 \)) \( \cdots \) (\( K_n \) \( E_m \))

\( \rightarrow \) \( Z_n^{m,k} \) \( E_0 \) \( \cdots \) \( E_m \) \( 0 \leq k \leq m \)

As we have done previously, we can show that this optimization holds by showing that both sides of the reduction are equivalent.
Therefore, we know that Opt 7 will indeed reduce the form we generated in Example 3 above. We will now examine Example 3 again in light of this new optimization rule.

Example 3

```
let a = 1
in
  let b = 2
  in
    a + b
```

Lambda calculus notation: \([\lambda a \ . \ [\lambda b \ . \ + a b] 2] 1\)

Abstraction:

\[
(\[a\]([b] + a b) 2) 1
\]
\[
(\[a\](S^2 (K_1 (+) (K_1 a) I)) 2) 1
\]

optimizing

\[
(\[a\](X^2_1 +a I) 2) 1
\]
\[
S^1_1 ([a] X^2_1 + a I)([a] 2) 1
\]
\[
S^1_1 (S^1_1 ([a] X^2_1 + a I)([a] (+)([a] a)([a] I))(K_1 2) 1
\]
\[
S^1_1 (S^1_1 (K_1 X^2_1 + a I)(K_1 I)) (K_1 2) 1
\]

optimizing
\[ S^1_1 (Z^{3,2}_1 X^{2,2}_1 + I I) (K_1 2) 1 \]


\[ Y_{1,1}^{1,1} (Z^{3,2}_1 X^{2,2}_1 + I I) 2 1 \]

Reduction:

\[ \rightarrow (Z^{3,2}_1 X^{2,2}_1 + I I) 2 \]
\[ \rightarrow X^{2,2}_1 + (I I) I 2 \]
\[ \rightarrow + (I 1) (I 2) \]
\[ \rightarrow + 1 2 \]
\[ \rightarrow 3 \]

Now, this last example points out that nested declarations are not handled by our new method as well as by Turner’s or Abdali’s procedures. Suppose we were to slightly modify the example to use simultaneous definitions of the variables. Example 4 follows through such an evaluation.

Example 4

\[ \text{let } a = 1 \text{ and } b = 2 \]
\[ \text{in } a + b \]

Lambda calculus notation: \[ [\lambda(a, b). + a b](1.2) \]

Abstraction:

\[ ([a, b] + a b)(1.2) \]
\[ S^2_2 ([a, b] +) ([a, b] a) ([a, b] b)(1.2) \]
\[ S^2_2 (K_2 +) I^2_2 I^2_2 (1.2) \]
\[ + (1.2) \]

This result is the same combinator expression as arrived at through Abdali’s method.

Turner’s method, as previously discussed, can not handle the simultaneous declarations.
3.4.1 More Examples

We present at this time a new group of examples, primarily designed to illustrate the handling of simultaneous declarations and multiple parameters.

Example 5

```
let a = 1 and b = 2 and c = 3
in
  a + b + c
```

Lambda calculus notation: \[\lambda(a,b,c). + (a+b)c\] (1,2,3)

Abstraction:

\[
\begin{align*}
S^2 & (\lambda(a,b,c). + (a+b)c) (1,2,3) \\
S^3 & (K_3 +) (\lambda(a,b,c). + (a+b)c) (1,2,3) \\
S^3 & (K_3 +) (S^3 ([a,b,c] +) ([a,b,c] a) ([a,b,c] b)) I^3_1 (1,2,3) \\
S^3 & (K_3 +) (S^3 (K_3 +) I^3_1 I^3_2) I^3_2 (1,2,3)
\end{align*}
\]

optimizing

\[
\begin{align*}
S^2 & (K_3 +) (X^{3,1} + I^3_1 I^3_2) I^3_3 (1,2,3) \\
X^{3,1} & + (X^{3,1} + I^3_1 I^3_2) I^3_2 (1,2,3)
\end{align*}
\]

Reduction:

\[
\begin{align*}
\rightarrow & + (X^{3,1} + I^3_1 I^3_2 (1,2,3)) (I^3_3 (1,2,3)) \\
\rightarrow & + (+ (I^3_1 (1,2,3)) (I^3_2 (1,2,3))) 3 \\
\rightarrow & + (1 2) 3 \\
\rightarrow & + 3 3 \\
\rightarrow & 6
\end{align*}
\]

We can compare this to Abdali's method:

Abstraction:

\[
\begin{align*}
([a,b,c] + (+ a b) c) (1,2,3) \\
B^3 & + ([a,b,c] + a b) ([a,b,c] c) (1,2,3)
\end{align*}
\]
Recall that we were able to replace the $B^m$ combinator optimization in our new method by the $X_n^{m,k}$ combinator optimization. Thus, in the new abstraction method we introduce the $X_n^{m,k}$ combinator rather than the $B^m$ combinator. In this case, they are equivalent, as the final results of our method and Abdali's method show.

Example 6

\[
\text{let } f (a,b) = a + b \\
\text{in } f ((3,4),1)
\]

Lambda calculus notation: \([\lambda f . f (3,4) 1] (\lambda (a,b) . a + b)\)

Abstraction:

\[
((f f (3 4) 1) ([a . b] + a b))
\]

\[
S^2 I (S^2 I (K_1 3) (K_1 4)) (K_1 1) +
\]

\[
Y_z^2 I (Z_z^0 I 3 4) 1 +
\]

Reduction:

\[
\rightarrow (I +) (Z_z^0 I 3 4 +) 1
\]

\[
\rightarrow (I +) 3 4 1
\]

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

\[
\rightarrow + 7 1
\]

\[
\rightarrow 8
\]

We can compare the above results to the combinator string produced by following Abdali's method.
We can see that our new method produced a combinator sequence with fewer combinators. The abstraction process took a little longer to complete, but the execution time will be faster since there are fewer combinators to reduce.

Example 7

let $f(x) = x + 3$ and $a = 1$
in
$$f(2)$$

Lambda calculus notation: $\lambda(f.a).f~2~(\lambda~x~.~x + 3.1)$

Abstraction:

$$(f~a)~f~2~([x] + x~3.1)$$

$S_2^1 ((f~a)~f) (f~a~2) (S_2^2 ([x] +) ([x] x)) ([x] 3.1)$$

$S_2^1 I_2^1 (K_1~2) (S_2^2 (K_1 +) I~(K_1~3))$$

$Y_2^{11} I_2^1 2 (Z_2^{11} + I~3.1)$$

Reduction:
Abdali's abstraction method produces the following combinator sequence:

Abstraction:

\[
\begin{align*}
&(\lambda x (\lambda a \, a \, 2) ([x] + x \, 3.1)) \Rightarrow \\
&B_1^2 \, I \, I_3^2 \, (\lambda f \, a \, f \, 2) (B_1^2 \, I \, ([x] \, x \, 3) \, 3) \Rightarrow \\
&B_1^2 \, I \, I_3^2 \, (K_2 \, 2) (B_1^2 \, I \, (K_1 \, 3) \, 1)
\end{align*}
\]

Reduction:

\[
\begin{align*}
&\Rightarrow I \, (I_2^2 \, B_1^2 \, I \, (K_1 \, 3), 1) \, (K_2 \, 2 \, (B_1^2 \, I \, (K_1 \, 3), 1) \, 1) \Rightarrow \\
&(I_2^2 \, B_1^2 \, I \, (K_1 \, 3), 1) \, 2 \Rightarrow \\
&B_1^2 \, I \, (K_1 \, 3), 2 \Rightarrow \\
&+ (I \, 2) (K_1 \, 3), 2 \Rightarrow \\
&+ 2 \, 3 \Rightarrow \\
&5
\end{align*}
\]

We can compare the number of combinators generated by each abstraction technique and see that our new method results in a smaller number of combinators to be reduced.

We, of course, are not going to prove that our method results in a smaller number of combinators. It seems to and our examples suggest that it generally will. This is a matter for future work. Our desire, at this time, is not to arrive at a best set of combinators, but rather to establish a framework within which various candidate combinator families can be studied.
3.5 Review of a Fixed-Program Machine

The abstraction methods presented in the previous section compile program code (lambda expressions) into combinator code. We have shown in many of the examples how the code is evaluated by using the reduction rules of the various combinators. This section will discuss a method of evaluation that was presented by Jones and Muchnick [10].

The method Jones and Muchnick present is a translation of combinator code into fixed program code. This fixed code is intended for execution by a stack machine. In addition to the compiler, they also have written an evaluator for the fixed code. The evaluator, written in LISP, executes the fixed code.

The translation of program code into target code is easily performed with a small set of compilation rules. The combinator expression is first represented as a binary tree. Each subtree represents a subexpression of the sequence, and is either a constant or a pair of subexpressions. As an example, consider the combinator sequence $C + 2 1$. This result is in Turner’s notation, obtained from the following example.

Example 3

```
let a = 1
in
let b = 2
in
a + b
```

Lambda calculus notation: $\lambda a . \left( \lambda b . + a b \right) 2 1$

Abstraction:

- $\left( \left[ a \right] \left( \left[ b \right] (+ a b ) 2 \right) 2 \right) 1$
- $\left( \left[ a \right] \left( S \left( \left[ b \right] + a \right) \left( \left[ b \right] b \right) \right) 2 \right) 1$
- $\left( \left[ a \right] \left( S \left( S \left( \left[ b \right] + \right) \left( \left[ b \right] a \right) \right) I ) 2 \right) 1$

In Jones and Muchnick's evaluation procedure, the expression \( C + 2 \) is first represented as a binary tree.

The translation of a combinator expression to stack machine code is made by replacing each subexpression by a labeled sequence of elementary actions. These actions include some directly related to the combinators (such as do_I, do_K, etc.) and other actions for evaluation purposes. We will discuss some of these later in this chapter. The compilation rules are listed in Appendix 1. Following the rules, we first label the expression tree. The numbering scheme we follow is to label the nodes sequentially within each level.
The following code is generated for the above tree:

```
(def example3
  (lambda ()
    (prog ()
      L0  (bind 'example 'L10)
           (look_up 'example)
           (first_call 'done)
      done (finish)
      L10 (push 'L30)
           (push 'L50)
           (push 'L70)
           (do_C)
           (do_top)
      L30 (push 1)
           (pexit)
      L50 (push 2)
           (pexit)
      L70 (call1 'L71)
      L71 (call2 'L72)
      L72 (do_plus)
           (pexit)))))
```

Once the code is generated, it can be evaluated by the LISP program that Jones and Muchnick presented. Their evaluator is also contained in Appendix 1. A detailed trace of the evaluation would be too tedious, but highlights of the process are of some interest.

Two stacks are set up for the evaluation process: stack and ret_stack. The stack is initialized to contain the label associated with the function being evaluated. In this case the example is bound to label L10. The return stack is set up with the terminating label 'done. The function that directs the work to be done in the evaluator is do_top. After setting up certain counters and the stack, the function first_call issues a call to do_top.

```
main_program example_3
  processors=1
  (stack= (L10))
  (ret_stack= (done))
  (do_top)
  (stack= nil)
  (ret_stack= (done))
  (push L30)
  (stack= (L30))
  (ret_stack= (done))
  (push L50)
  (stack= (L50 L30))
```
(ret_stack = (done))
(push L70)
(stack = (L70 L50 L30))
(ret_stack = (done))
(do_C)
(stack = (L70 L30 L50))
(ret_stack = (done))
(do_top)
(stack = (L30 L50))
(ret_stack = (done))
(old_call1 L71)
(stack = (L30 oldcall))
(ret_stack = (L71 (L50) done))
(do_top)
(stack = (oldcall))
(ret_stack = (L71 (L50) done))
(push 1)
(stack = (1 oldcall))
(ret_stack = (L71 (L50) done))
(pexit)
(stack = (1))
(ret_stack = (L71 (L50) done))
(old_call2 L72)
(stack = (L50 oldcall))
(ret_stack = (L72 1 nil done))
(do_top)
(stack = (oldcall))
(ret_stack = (L72 1 nil done))
(push 2)
(stack = (2 oldcall))
(ret_stack = (L72 1 nil done))
(pexit)
(stack = (2))
(ret_stack = (L72 1 nil done))
(old_do_plus)
(stack = (3))
(ret_stack = (done))
(pexit)

As we would expect, the result is left on the stack.

A few words of explanation about the instructions and their effect are in order at this time. The code at L70 is (call1 'L71). The action of call1 is to obtain the value of the first operand of a binary operation. The L71 acts as a return address. For reasons that will become clear later, when this example (and other examples, as well) is executed on a single processor system, it is necessary to flag the value with a tag of oldcall. Similarly, call2 is used to obtain the value of the second operand, and it, too, must have a flag of oldcall. The
operation (do_top) acts as a sequencer through the code, while (pexit) acts as a subroutine return.

3.6 Modification of the Evaluator

The compiled approach, such as presented by Jones and Muchnick, interests us because the code can easily be evaluated by a conventional computer. Other benefits from using this method include a simpler storage management technique (over that required, for example, by Turner or Clarke (SKIM)), and the efficiency of compiled code (as compared to interpreted code).

The evaluation mechanism can be modified to accommodate a different set of combinators. We have introduced changes to this evaluator to handle the extended set of combinators discussed in the last section. In particular, the combinator families $K_n$, $I_n^m$, $S_n^m$, $X_n^{m,k}$, $Y_n^{m,k}$, and $Z_n^{m,k}$ are supported by the new evaluator. The evaluator for our set of combinators is contained in Appendix 2.

Modifications to the compiler must allow for the change from handling single combinators to using families of combinators. In particular, rule 2 must change to record the parameters of our combinators. The revised rule is as follows.
2a. Code \( \left( p \right) = L_p \left( \text{do}_K n \right) \)
\[
\begin{align*}
K_n & \quad \text{(do\_top)} \\
\end{align*}
\]

2b. Code \( \left( p \right) = L_p \left( \text{do}_* m n \right) \)
\[
\begin{align*}
*_{m/n} & \quad \text{(do\_top)} \\
\end{align*}
\]
where * is either I or S

2c. Code \( \left( p \right) = L_p \left( \text{do}_* m n k \right) \)
\[
\begin{align*}
*_{m/n/k} & \quad \text{(do\_top)} \\
\end{align*}
\]
where * is any of the combinators X, Y, or Z

2d. Code \( \left( p \right) = L_p \left( \text{do}_I \right) \)
\[
\begin{align*}
I & \quad \text{(do\_top)} \\
\end{align*}
\]

The rest of the compilation rules remain the same (as listed in Appendix 1).

The evaluator must also be modified to reduce expressions containing the new combinator families. Specifically, new definitions for do_S and do_K are needed, as well as introducing do_newI, do_X, do_Y, and do_Z. These definitions are contained in Appendix 2.

3.6.1 Evaluation of Examples Using the Modified Evaluator

The following examples will show that the method proposed by Jones and Muchnick can be adapted to handle the new combinator set.

Example 3

\[
\text{let } a = 1 \\
\quad \text{in} \\
\text{let } b = 2 \\
\quad \text{in} \\
\quad a + b
\]

Combinator expression: \( Y_1^{1.1} \left( Z_1^{1.2} \right) X_1^{2.2} + I I \) 2 1
Generated code:

```
(def ex3
  (lambda ()
    (prog ()
      (bind 'example3 'L10)
      (look_up 'example3)
      (first_call 'done)
      done (finish)
      L10 (push 'L30)
        (push 'L50)
        (push 'L70)
        (do_Y 1 1 1)
        (do_top)
      L30 (push 1)
        (pexit)
      L50 (push 2)
        (pexit)
      L70 (push 'L90)
        (push 'L110)
        (push 'L130)
        (push 'L150)
        (do_Z 3 1 2)
        (do_top)
      L90 (do_1)
        (do_top)
      L110 (do_1)
        (do_top)
      L130 (call1 'L131)
      L131 (call2 'L132)
      L132 (do_plus)
        (pexit)
      L150 (do_X 2 1 2)
        (do_top))))
```

An evaluation trace follows.

```
main_program Example_3
  processors=1
  (stack= (L10))
  (ret_stack= (done))
  (do_top)
  (stack= nil)
  (ret_stack= (done))
  (push L30)
  (stack= (L30))
  (ret_stack= (done))
  (push L50)
  (stack= (L50 L30))
  (ret_stack= (done))
  (push L70)
```
(stack= (L70 L50 L30))
(ret_stack= (done))
(do_Y 1 1 1)
(stack= ((L70 L30) L50))
(ret_stack= (done))
(do_top)
(stack= (L30 L50))
(ret_stack= (done))
(push L90)
(stack= (L90 L30 L50))
(ret_stack= (done))
(push L110)
(stack= (L110 L90 L30 L50))
(ret_stack= (done))
(push L130)
(stack= (L130 L110 L90 L30 L50))
(ret_stack= (done))
(push L150)
(stack= (L150 L130 L110 L90 L30 L50))
(ret_stack= (done))
(do_Z 3 1 2)
(stack= (L150 L130 (L110 L30) L90 L50))
(ret_stack= (done))
(do_top)
(stack= (L130 (L110 L30) L90 L50))
(ret_stack= (done))
(do_X 2 1 2)
(stack= (L130 (L110 L30) (L90 L50)))
(ret_stack= (done))
(do_top)
(stack= ((L110 L30) (L90 L50)))
(ret_stack= (done))
(old_call1 L131)
(stack= ((L110 L30) oldcall))
(ret_stack= (L131 ((L90 L50)) done))
(do_top)
(stack= (L30 oldcall))
(ret_stack= (L131 ((L90 L50)) done))
(do_J)
(stack= (L30 oldcall))
(ret_stack= (L131 ((L90 L50)) done))
(do_top)
(stack= (oldcall))
(ret_stack= (L131 ((L90 L50)) done))
(push 1)
(stack= (1 oldcall))
(ret_stack= (L131 ((L90 L50)) done))
(exit)
(stack= (1))
(ret_stack= (L131 ((L90 L50)) done))
(old_call2 L132)
(stack= ((L90 L50) oldcall))
(ret_stack= (L132 1 nil done))
(do_top)
  (stack= (L50 oldcall))
  (ret_stack= (L132 1 nil done))
(d0_1)
  (stack= (L50 oldcall))
  (ret_stack= (L132 1 nil done))
(do_top)
  (stack= (oldcall))
  (ret_stack= (L132 1 nil done))
(push 2)
  (stack= (2 oldcall))
  (ret_stack= (L132 1 nil done))
pexit)
  (stack= (2))
  (ret_stack= (L132 1 nil done))
(old_do_plus)
  (stack= (3))
  (ret_stack= (done))
pexit)

Example 4

let a = 1 and b = 2
in a+b

Combinator expression: + 1 2

Generated code:

(def ex4)
  (lambda ()
    (prog ()
      (bind 'example4 'L10)
      (look_up 'example4)
      (first_call 'done)
      done (finish)
      L10 (push 'L30)
      (push 'L50)
      (call1 'L41)
      L41 (call2 'L42)
      L42 (do_plus)
      (pexit)
      L30 (push 2)
      (pexit)
      L50 (push 1)
      (pexit)))

The following sequence of evaluation steps occur:
Example 4

```plaintext
main_program Example_4
processors=1
(stack= (L10))
(ret_stack= (done))
(do_top)
(stack= nil)
(ret_stack= (done))
(push L30)
(stack= (L30))
(ret_stack= (done))
(push L50)
(stack= (L50 L30))
(ret_stack= (done))
(old_call1 L41)
(stack= (L50 oldcall))
(ret_stack= (L41 (L30) done))
(do_top)
(stack= (oldcall))
(ret_stack= (L41 (L30) done))
(push 1)
(stack= (1 oldcall))
(ret_stack= (L41 (L30) done))
(pexit)
(stack= (1))
(ret_stack= (L41 (L30) done))
(old_call2 L42)
(stack= (L30 oldcall))
(ret_stack= (L42 1 nil done))
(do_top)
(stack= (oldcall))
(ret_stack= (L42 1 nil done))
(push 2)
(stack= (2 oldcall))
(ret_stack= (L42 1 nil done))
(pexit)
(stack= (2))
(ret_stack= (L42 1 nil done))
(old_do_plus)
(stack= (3))
(ret_stack= (done))
(pexit)
```

Example 5

```plaintext
let a = 1 and b = 2 and c = 3
in
  a + b + c
```

Combinator expression: \( X_2 \cdot (X_2 \cdot (1 \cdot 2) \cdot (3) \cdot (1 \cdot 2 \cdot 3)) \)
(def ex5
  (lambda ()
    (prog ()
      (bind 'example5 'L10)
      (look_up 'example5)
      (first_call 'done)
      (do_top)
      (finish)
      (do_X 2 3 1)
      (do_top)
      (L10 (push 'L30))
      (push 'L50)
      (push 'L70)
      (push 'L90)
      (push 'L110)
      (push 'L130)
      (do_X 2 3 1)
      (do_top)
      (L30 (push 3))
      (pexit)
      (L50 (push 2))
      (pexit)
      (L70 (push 1))
      (pexit)
      (L90 (do_newl 3 3))
      (do_top)
      (L110 (push 'L150))
      (push 'L170)
      (push 'L190)
      (do_X 2 3 1)
      (do_top)
      (L130 (call1 'L131))
      (call2 'L132)
      (L132 (do_plus))
      (pexit)
      (L150 (do_newl 2 3))
      (do_top)
      (L170 (do_newl 1 3))
      (do_top)
      (L190 (call1 'L191))
      (call2 'L192)
      (L192 (do_plus))
      (pexit))))

-Evaluation steps:

main_program  Example_5
  processors=1
  (stack= (L10))
  (ret_stack= (done))
  (do_top)
  (stack= nil)
  (ret_stack= (done))
  (push L30)
  (stack= (L30))
(ret_stack= (done))
(push L50)
(stack= (L50 L30))
(ret_stack= (done))
(push L70)
(stack= (L70 L50 L30))
(ret_stack= (done))
(push L90)
(stack= (L90 L70 L50 L30))
(ret_stack= (done))
(push L110)
(stack= (L110 L90 L70 L50 L30))
(ret_stack= (done))
(push L130)
(stack= (L130 L110 L90 L70 L50 L30))
(ret_stack= (done))
(do_X 2 3 1)
(stack= (L130 (L110 L70 L50 L30) (L90 L70 L50 L30)))
(ret_stack= (done))
(do_top)
(stack= ((L110 L70 L50 L30) (L90 L70 L50 L30)))
(ret_stack= (done))
(old_call L131)
(stack= ((L110 L70 L50 L30) oldcall))
(ret_stack= (L131 ((L90 L70 L50 L30)) done))
(do_top)
(stack= (L70 L50 L30 oldcall))
(ret_stack= (L131 ((L90 L70 L50 L30)) done))
(push L150)
(stack= (L150 L70 L50 L30 oldcall))
(ret_stack= (L131 ((L90 L70 L50 L30)) done))
(push L170)
(stack= (L170 L150 L70 L50 L30 oldcall))
(ret_stack= (L131 ((L90 L70 L50 L30)) done))
(push L190)
(stack= (L190 L170 L150 L70 L50 L30 oldcall))
(ret_stack= (L131 ((L90 L70 L50 L30)) done))
(do_X 2 3 1)
(stack= (L190 (L170 L70 L50 L30) (L150 L70 L50 L30) oldcall))
(ret_stack= (L131 ((L90 L70 L50 L30)) done))
(do_top)
(stack= ((L170 L70 L50 L30) (L150 L70 L50 L30) oldcall))
(ret_stack= (L131 ((L90 L70 L50 L30)) done))
(old_call L191)
(stack= ((L170 L70 L50 L30) oldcall))
(ret_stack= (L191 ((L150 L70 L50 L30) oldcall) L131 ((L90 L70 L50 L30)) done))
(do_top)
(stack= (L70 L50 L30 oldcall))
(ret_stack= (L191 ((L150 L70 L50 L30) oldcall) L131 ((L90 L70 L50 L30)) done))
(do_new 1 3)
(stack= (L70 oldcall))
(ret_stack= (L191 ((L150 L70 L50 L30) oldcall) L131 ((L90 L70 L50 L30)) done))
(do_top)
(stack = (oldcall))
(ret_stack = (L191 ((L150 L70 L50 L30) oldcall) L131 ((L90 L70 L50 L30) done))
(push 1)
(stack = (1 oldcall))
(ret_stack = (L191 ((L150 L70 L50 L30) oldcall) L131 ((L90 L70 L50 L30) done))
(pexit)
(stack = (1))
(ret_stack = (L191 ((L150 L70 L50 L30) oldcall) L131 ((L90 L70 L50 L30) done))
(old_call2 L192)
(stack = ((L150 L70 L50 L30) oldcall))
(ret_stack = (L192 1 (oldcall) L131 ((L90 L70 L50 L30) done))
(do_top)
(stack = (L70 L50 L30 oldcall))
(ret_stack = (L192 1 (oldcall) L131 ((L90 L70 L50 L30) done))
(do_newl 2 3)
(stack = (L50 oldcall))
(ret_stack = (L192 1 (oldcall) L131 ((L90 L70 L50 L30) done))
(do_top)
(stack = (oldcall))
(ret_stack = (L192 1 (oldcall) L131 ((L90 L70 L50 L30) done))
(push 2)
(stack = (2 oldcall))
(ret_stack = (L192 1 (oldcall) L131 ((L90 L70 L50 L30) done))
(pexit)
(stack = (2))
(ret_stack = (L192 1 (oldcall) L131 ((L90 L70 L50 L30) done))
(old_do_plus)
(stack = (3 oldcall))
(ret_stack = (L131 ((L90 L70 L50 L30) done))
(pexit)
(stack = (3))
(ret_stack = (L131 ((L90 L70 L50 L30) done))
(old_call2 L132)
(stack = ((L90 L70 L50 L30) oldcall))
(ret_stack = (L132 3 nil done))
(do_top)
(stack = (L70 L50 L30 oldcall))
(ret_stack = (L132 3 nil done))
(do_newl 3 3)
(stack = (L30 oldcall))
(ret_stack = (L132 3 nil done))
(do_top)
(stack = (oldcall))
(ret_stack = (L132 3 nil done))
(push 3)
(stack = (3 oldcall))
(ret_stack = (L132 3 nil done))
(pexit)
(stack = (3))
(ret_stack = (L132 3 nil done))
(old_do_plus)
(stack = (6))
(ret_stack = (done))
Example 6

\[ \text{let } f \ (a, b) = a + b \]
\[ \text{in } f \ (f \ (3, 4).1) \]

Combinator expression: \( Y I^2 (Z I^0 \ 1 \ 3 \ 4 \ 1) \)

Generated code:

\[
\begin{align*}
(\text{def ex6}) & \\
& (\text{lambda} () ) \\
& (\text{prog} () ) \\
& \quad (\text{bind} \ 'example6 \ 'L10) \\
& \quad (\text{look_up} \ 'example6) \\
& \quad (\text{first_call} \ 'done) \\
& \quad \text{done} \ (\text{finish}) \\
& \quad \text{L10} \ (\text{push} \ 'L30) \\
& \quad \quad (\text{push} \ 'L50) \\
& \quad \quad (\text{push} \ 'L70) \\
& \quad \quad (\text{push} \ 'L90) \\
& \quad \quad (\text{do_Y} \ 2 \ 1 \ 2) \\
& \quad \quad (\text{do_top}) \\
& \quad \text{L30} \ (\text{call1} \ 'L31) \\
& \quad \text{L31} \ (\text{call1} \ 'L32) \\
& \quad \text{L32} \ (\text{do_plus}) \\
& \quad \quad (\text{peexit}) \\
& \quad \text{L50} \ (\text{push} \ 1) \\
& \quad \quad (\text{peexit}) \\
& \quad \text{L70} \ (\text{push} \ 'L110) \\
& \quad \quad (\text{push} \ 'L130) \\
& \quad \quad (\text{push} \ 'L150) \\
& \quad \quad (\text{do_Z} \ 2 \ 1 \ 0) \\
& \quad \quad (\text{do_top}) \\
& \quad \text{L90} \ (\text{do_1}) \\
& \quad \quad (\text{do_top}) \\
& \quad \text{L110} \ (\text{push} \ 4) \\
& \quad \quad (\text{peexit}) \\
& \quad \text{L130} \ (\text{push} \ 3) \\
& \quad \quad (\text{peexit}) \\
& \quad \text{L150} \ (\text{do_1}) \\
& \quad \quad (\text{do_top})) \\
\end{align*}
\]

Evaluation steps:

main_program Example 6
   processors=1
   (stack= (L10))
   (ret_stack= (done))
(do_top)
  (stack= nil)
  (ret_stack= (done))
(push L30)
  (stack= (L30))
  (ret_stack= (done))
(push L50)
  (stack= (L50 L30))
  (ret_stack= (done))
(push L70)
  (stack= (L70 L50 L30))
  (ret_stack= (done))
(push L90)
  (stack= (L90 L70 L50 L30))
  (ret_stack= (done))
(do Y 2 1 2)
  (stack= ((L90 L30) (L70 L30) L50))
  (ret_stack= (done))
(do_top)
  (stack= (L30 (L70 L30) L50))
  (ret_stack= (done))
(do I)
  (stack= (L30 (L70 L30) L50))
  (ret_stack= (done))
(do_top)
  (stack= ((L30 L70 L30) L50))
  (ret_stack= (done))
(old_call L31)
  (stack= ((L70 L30) oldcall))
  (ret_stack= (L31 (L50) done))
(do_top)
  (stack= (L30 oldcall))
  (ret_stack= (L31 (L50) done))
(push L110)
  (stack= (L110 L30 oldcall))
  (ret_stack= (L31 (L50) done))
(push L130)
  (stack= (L130 L110 L30 oldcall))
  (ret_stack= (L31 (L50) done))
(push L150)
  (stack= (L150 L130 L110 L30 oldcall))
  (ret_stack= (L31 (L50) done))
(do Z 2 1 0)
  (stack= ((L150 L30) L130 L110 oldcall))
  (ret_stack= (L31 (L50) done))
(do_top)
  (stack= (L30 L130 L110 oldcall))
  (ret_stack= (L31 (L50) done))
(do I)
  (stack= (L30 L130 L110 oldcall))
  (ret_stack= (L31 (L50) done))
(do_top)
  (stack= (L130 L110 oldcall))
(ret_stack= (L31 (L50) done))
(old_call1 L31)
(stack= (L130 oldcall))
(ret_stack= (L31 (L110 oldcall) L31 (L50) done))
(do_top)
(stack= (oldcall))
(ret_stack= (L31 (L110 oldcall) L31 (L50) done))
(push 3)
(stack= (3 oldcall))
(ret_stack= (L31 (L110 oldcall) L31 (L50) done))
(pexit)
(stack= (3))
(ret_stack= (L31 (L110 oldcall) L31 (L50) done))
(old_call2 L32)
(stack= (L110 oldcall))
(ret_stack= (L32 3 (oldcall) L31 (L50) done))
(do_top)
(stack= (oldcall))
(ret_stack= (L32 3 (oldcall) L31 (L50) done))
(push 4)
(stack= (4 oldcall))
(ret_stack= (L32 3 (oldcall) L31 (L50) done))
(pexit)
(stack= (4))
(ret_stack= (L32 3 (oldcall) L31 (L50) done))
(old_do_plus)
(stack= (7 oldcall))
(ret_stack= (L31 (L50) done))
(pexit)
(stack= (7))
(ret_stack= (L31 (L50) done))
(old_call2 L32)
(stack= (L50 oldcall))
(ret_stack= (L32 7 nil done))
(do_top)
(stack= (oldcall))
(ret_stack= (L32 7 nil done))
(push 1)
(stack= (1 oldcall))
(ret_stack= (L32 7 nil done))
(pexit)
(stack= (1))
(ret_stack= (L32 7 nil done))
(old_do_plus)
(stack= (8))
(ret_stack= (done))
(pexit)
3.7 Multiprocessor Support

We have investigated the possibility of a multiprocessor evaluation of fixed code programs, such as the programs generated via the Jones' technique. Our studies have indicated that the combinator functions can be used effectively as the mechanisms to direct work to subprocessors. To do so, we divide the support of multiprocessing features into three parts:

1. Allocation of processors
2. Synchronization of execution
3. Deallocation of processors

3.7.1 Allocation of Processors

The combinators dictate what portion of code is to be allocated to a child processor. After a combinator "reduction" is performed, a modified stack results. Using the parameters to the combinator, we are able to decompose this resulting stack and allocate new processors to evaluate portions of it. A detailed description of this allocation procedure for each combinator follows.

\[ S_n^n \times y_1 \cdots y_m z_1 \cdots z_n \rightarrow (x \ z_1 \cdots z_n) (y_1 \ z_1 \cdots z_n) \cdots (y_m \ z_1 \cdots z_n) \]

The \( S_n^n \) combinator creates \( m+1 \) elements on the stack. Provided there are enough processors, each of the elements 2 through \( m+1 \) are used as initial stack contents for \( m \) new processors. The current processor's stack is marked in an appropriate manner to indicate that these portions are being evaluated by a subprocessor. The current (parent) processor continues execution and evaluates the first stack element. Eventually the results from the \( m \) new processors will be required to continue further evaluation in the current processor. The necessary synchronization between the current processor and the subprocessors it allocated will be discussed in a subsequent section.
To illustrate this allocation of processors, suppose we have the following:

\[ \text{stack} = (L\ 270\ L\ 170\ L\ 110\ L\ 70\ L\ 50\ L\ 30) \]

and the next instruction to be executed is (do_newS 3 2). The resulting stack is

\[ \text{stack} = ((L\ 270\ L\ 50\ L\ 30)\ (L\ 170\ L\ 50\ L\ 30)\ (L\ 110\ L\ 50\ L\ 30)\ (L\ 70\ L\ 50\ L\ 30)) \]

Since \( m = 3 \), there are 4 items resulting on the stack. Each of these items can be evaluated by a different processor. Suppose the current processor is called processor 1. We can then start up processor 2 with an initial stack of \((L\ 170\ L\ 50\ L\ 30)\), processor 3 with an initial stack of \((L\ 110\ L\ 50\ L\ 30)\), and processor 4 with an initial stack of \((L\ 70\ L\ 50\ L\ 30)\). The stack of the current processor is then modified to indicate that these processors have been started up to evaluate certain elements of the stack.

\[ \text{stack} = ((L\ 270\ L\ 50\ L\ 30)\ (\text{proc}\ 2)\ (\text{proc}\ 3)\ (\text{proc}\ 4)) \]

Processor 1 continues execution by evaluating the \((L\ 270\ L\ 50\ L\ 30)\), until the results from processor 2, processor 3, or processor 4 are required to continue evaluation. The section on synchronization of processors will discuss this in greater detail.

\[ X_n^{m,k} y_0 \cdots y_m x_1 \cdots x_n \rightarrow y_0 \cdots y_{k-1} (y_k x_1 \cdots x_n) \cdots (y_m x_1 \cdots x_n) \]

The number of processors initiated after the execution of an \( X_n^{m,k} \) combinator depends on both the \( m \) and \( k \) parameters to \( X \). If \( k = 0 \), then \( m \) processors are allocated. The first element of the stack is not sent as an initial stack to a new processor, but is evaluated by the current processor when execution resumes. The next \( m \) elements (elements 2 through \( m+1 \)) are allocated to new processors. When \( k \neq 0 \), then processors are allocated for the \( k+1 \) element through the \( m+1 \) element of the stack.

\[ Y_n^{m,k} y_0 \cdots y_m x_1 \cdots x_n \rightarrow (y_0 x_1 \cdots x_n) \cdots (y_{k-1} x_1 \cdots x_n) y_k \cdots y_m \]

The allocation scheme calls for \( m \) processors to be allocated after a \( Y_n^{m,k} \) combinator is evaluated. As in each case above, the first element of the stack is not allocated for evaluation, but remains on the stack for the current processor to evaluate. The remaining elements, 2 through \( m+1 \), are used as initial stack values for \( m \) new processors.
After a $Z_{n}^{m,k}$ combinator has completed rearranging the stack, the allocation of processors follows the same scheme as the $X_{n}^{m,k}$ combinator. If $k = 0$ then processors are allocated for elements 2 through $m + 1$. If $k \neq 0$ then processors are initiated with the $k + 1$ element through the $m + 1$ element of the stack.

$$K_{n} \ a \ b_{1} \cdots \ b_{n} \rightarrow a$$

Since evaluation of the $K_{n}$ combinator results in only one element, there is no need to allocate new processors to evaluate this element. The current processor continues evaluation of $a$.

$$I_{n}^{m} \ a_{1} \cdots \ a_{n} \rightarrow a_{m}$$

The $I_{n}^{m}$ combinator also produces a single element, therefore no new processors are allocated in this case either.

$$I \ x \rightarrow x$$

The $I$ combinator, actually just a shorthand notation for $I_{1}^{1}$, results in a single element that can be evaluated by the current processor.

In general, the combinatorstake care of the majority of processor generation. There do arise other circumstances when processors may be started. These circumstances exist during execution of a binary operator. When it comes time to actually perform the operation, both operands must be on the top of the stack. More importantly, if either operand is not in a final numeric form, new processors can be started to evaluate the operands. The initial stack for the new processor(s) is contained in either the first element of the stack or the second element of the stack, or both elements if neither operand is in a final numeric form.
3.7.2 Synchronization of Execution

As in any multiprocessor arrangement, there must be some synchronization between processors. The only communication required between processors in our proposed model is the passing of the initial stack from the parent processor to a child processor and the return stack from the child to the parent. In our terminology, a parent processor is one that allocated a portion of its own stack to be evaluated by another processor. This subprocessor is frequently called a child processor. A child processor has a copy of the fixed program code (all processors have this code in local memory). Each processor works with its own stack and memory, independent of the other processors in the system, except for the possibility of allocating some of its work to other processors. Upon completion of the designated task, the child processor reports its results back to its parent. The parent processor has a record of the allocation of code to the child and makes the appropriate substitution of the child’s results back into its own stack. This process will be further described in the next section on deallocation of processors.

We have already described the algorithm we follow to decide when to allocate portions of the stack to children processors. In that discussion, we alluded to marking the stack. This marking is necessary for the parent processor to be able to recognize when work is being done by another processor and when the results from the child are required before any further evaluation can take place by the parent. The marking system creates what is called a *hole* in the stack in place of the code that was allocated to a child processor. We have chosen to illustrate this hole by a unique identification, the processor number. Since the system assumes a sufficient number of processors to complete the task, we can use the processor number to mark the stack, and in this way allow the parent processor to know which child’s result belongs at a particular marked place in its stack. The example we looked at in the above section illustrates this marking system.
The current processor can allocate each of the elements two through four to child processors, marking the stack to indicate this action. The parent processor needs to transmit the appropriate element to the child processor. The child processor receives this information and initializes its own stack with it.

\[ \text{stack} = (\langle L\ 270\ L\ 50\ L\ 30 \rangle \ \langle L\ 170\ L\ 50\ L\ 30 \rangle \ \langle L\ 110\ L\ 50\ L\ 30 \rangle \ \langle L\ 70\ L\ 50\ L\ 30 \rangle) \]

As the current processor continues evaluation, it may find that one or more of the results is required in order to continue evaluation. At this time, the processor must enter a busy wait until the required result(s) is available.

3.7.3 Deallocation of Processors

Once a processor has completed its task, it must report its results back to the processor that invoked it. These results may be a simple element, such as a numeric value, a list of numeric values, or even unevaluated code. This unevaluated code may result if the information sent by the parent processor was not sufficient to evaluate to a terminal condition. Such a situation arises when the parameters to a combinator are contained in the next element(s) of the parent's stack. This does not pose a problem. The child processor evaluates as much code as possible. If during evaluation, the child processor recognizes that there are not enough parameters for a specific operation (such as a combinator or a binary operation), the processor prepares to return its stack to the parent. This stack may or may not be different from the initial stack sent to the child processor. If the child was able to perform some work, then the stack will contain partially evaluated code. If the child allocated work to other processors, and the stack contains holes, we have decided to force the child to wait for the holes to fill before sending the results back to the parent. A parent will never receive results containing holes from a child processor.
When a processor recognizes that it can do no more work, either because of finding a terminal value on the top of its stack, or being short of parameters, its stack is returned to the parent processor. Once the results are reported to the parent, the child processor is deallocated and returned to the pool of free processors. The parent processor, upon receiving the results (which are uniquely identified), fills the appropriate hole in the stack. If the parent had been idle and waiting for the results, it can now continue evaluation. If the parent had not been waiting for the results, the hole is filled and will be used later during evaluation.
4. IMPLEMENTATION NOTES

After testing our proposed method with paper and pencil, we decided that the next step in evaluation would be to implement our ideas. We chose to do this by extending the fixed program machine concept to handle a number of processors, rather than to function only as a single processor sequential machine. In this way, we could simulate a true multiprocessor implementation. This chapter discusses the architecture of a system that supports this extended notion of a fixed program machine.

Our fixed program machine model includes one stack to handle the current computation and a second stack (ret_stack) to accommodate subroutine calls. An environment is also established to map user-defined functions to their code. However, this environment is merely an implementation technique for functional bindings and is not used to map bound variables to their values.

4.1 Machine Instructions

The instructions of our fixed program machine can be grouped into four different categories.

1. The combinator support instructions are primarily stack manipulation procedures. Each such instruction is named for the combinator it supports. For example, each of the combinators we introduced in a previous chapter has a supporting instruction. The action of the combinator $S_n^m$ is accomplished with the do_newS instruction and $X_{n^m}^k$ is supported by do_X. Similarly, we have included the instructions do_Y, do_Z, do_newI, do_newK, and do_I, as well as the instructions that support Turner's combinators (do_S, do_K, do_I, do_B, and do_C).
2. The basic instructions perform the basic operations such as addition. Included in this group are do_plus, do_times, do_minus, and do_eq. Each of these can be used in our multiprocessor system to perform the desired operation. However, if the system is running with a finite number of processors and not enough processors are available to take advantage of the parallelism that binary operands introduce, then there are complementary instructions to perform these operations. These complementary instructions are structured such that the entire operation of evaluating the operands and performing the operation is executed by a single processor. These single processor instructions are old_do_plus, old_do_minus, old_do_times, and old_do_eq. We are not satisfied with this approach of handling binary operators with two different sets of instructions, but are willing to work with it for now. Future plans include a modification in this area. Also included in this group of instructions are call_cond and do_cond which are available to handle conditionals in our programs.

3. The program code that is generated may also contain instructions that are control oriented. These control functions include do_top and pexit. The do_top instruction acts as a sequencer through the code. (Jones and Muchnick [10] call it a dispatcher.) The evaluation of operands is considered to be a subroutine call, supported by the control instructions call1 and call2. These instructions initiate argument evaluation of the operands of all binary operators. The function pexit performs the subroutine return.

4. The remaining instructions are miscellaneous in nature. These include start, finish, and first_call, which perform the expected tasks of program initiation and termination. As mentioned above, an environment exists to bind user defined functions to their code. This is handled by the instructions bind and look_up.
4.2 Multiprocessor Support

As we have discussed in the previous chapter, multiprocessing support consists of three parts: allocation of processors, synchronization of processors, and deallocation of processors. The duties required by each of these are handled by various supporting functions in our fixed program machine. These supporting functions will now be discussed.

We have demonstrated how the combinators direct the allocation of work to subprocessors. Depending on which combinator function is performed, various portions of the stack can be sent to available processors. These processors then complete their assigned work and report the results back to the initiating parent processor. Each of the combinator functions (do_newS, do_X, etc.) that is able to allocate portions of the stack to other processors contains a call to the function startprocs.

The startprocs routine and two associated functions, start_arg and fork, take care of initiating available processors with the "code" from the stack and marking the stack once the code is shipped to another processor for execution. The fork routine manages some of the necessary data structures that simulate our multiprocessing system, such as creating new stacks for the subprocessors, reporting unique ids for each subprocessor, and keeping track of the number of subprocessors created.

Since in functional languages we can evaluate operands in any order, we can evaluate them in parallel. This allows us to take advantage of potential parallelism in evaluating the operands. Each call to an operator contains a subroutine call to a function that may allocate the evaluation of the operands to child processors. This subroutine uses the fork function to handle the management of the child processors just as the combinator functions use fork.
The synchronization required between parent and child processors consists of:

1. the passing of the initial stack to the child,
2. the marking of the parent stack with a hole to indicate work is being executed by another processor,
3. the return of the child's results, and
4. the replacement of the hole in the parent stack by these results.

We choose to provide these functions in our multiprocessing system via the exchanging of "messages" between processors. We have included various mechanisms for this in our simulated system.

The fork procedure takes care of identifying each child processor with a unique id. This id is used as the marking identification within the parent stack. In a message passing system, this id could be passed back along with the results from the child and thereby identify where the results belong in the parent's stack. The fork routine also initializes the child processor's stack with the correct code from the parent.

Once a parent has forked off child processors to evaluate some of its code, the parent can continue execution. If, however, the parent requires the results from a child processor in order to continue processing, the parent must perform a "busy" wait until these results become available. Our combinator functions have been modified to simulate such a busy wait. Each of the binary operators has also been written to handle the situation where one or more of the operands is not available, due to processing by a child processor.

The remainder of the synchronization tasks are handled by a procedure called join. The job of the join function is to undo all the structures set up by the fork procedure and to modify the parent's stack to reflect the results reported by the child. Once a child processor has executed a join, the processor is no longer active and is once again made a part of the pool of available processors.
4.3 Management of a Finite Processor System

We initially designed our system for an infinite number of processors in order to take advantage of as much parallelism as possible. We then enhanced our system to model a setup with a variable number of processors. To achieve this flexibility, we needed to modify our fixed program machine to ask for an available processor when it wants to allocate work. If no processor is available, the requestor just ends up doing all of its own work rather than allocating portions of it to other processors.

This enhancement required a small change in the function start_arg. If there is an available processor, then this processor is given work to do; otherwise, the current processor performs the work just as in the single processor sequential execution. The current processor never really needs to know whether or not it has allocated work to be done elsewhere. After indicating those portions of its stack that can be allocated to other processors, it just proceeds to execute. If the allocation procedures found available processors for the work, the current processor may find a hole in its stack, in which case it must wait until a result replaces the hole. It may, of course, complete its execution task without ever having to wait for results. In this case, the processor can not tell if work was allocated to other processors or if it did all the work itself.

Our handling of the binary operators did cause a slightly more complicated solution for the modification of these functions when using a finite number of processors. We really would like to take advantage of as much parallelism as possible and start a processor to evaluate the first operand and one to evaluate the second operand. However, if only one processor is available, our algorithm may not handle using this one available processor to evaluate the first parameter. Part of the problem exists because the single processor machine uses the ret_stack for saving return addresses, saving values of first operands, and saving actual code that needs to be executed. Our multiprocessor machine does not utilize the ret_stack in this way, and the binary operations had been modified to reflect these
changes. For example, do_plus in the multiprocessor case expects its evaluated operands to be the first two elements of the stack. The single processor solution uses subroutines to evaluate the operands for the plus, and the old_do_plus expects one operand in the return stack and the other on the top of the stack. Thus, if one processor were available and the first operand were evaluated, its value would be found on the stack. Meanwhile, if the parent processor finds that it needs to evaluate the second parameter, it must revert to the single processor solution. When the addition is to be performed, we don't know where to find the operands (or whether the ret_stack contains information).

To solve this problem, it is necessary to have enough processors free (at least two) to evaluate both operands in parallel, or else we force the current processor to do the job in a single processor mode. Thus, if there is one free processor when a call1 is executed, the current processor will evaluate the call1, call2, do_bin_op, and pexit without any additional help. This does not mean that the one available processor can't be used. There do exist situations in combinator functions where a single processor can be given work to do. If two or more processors are available when the call1 is executed, the evaluation of both operands will be allocated to two separate processors and the results sent back to the parent. We realize this approach may be inefficient in some situations, but it provided us with a solution for the present. We intend to develop a better solution to this problem in the future.

4.4 System Requirements

Our proposed multiprocessor system consists of a collection of identical processors. Each processor executes those machine instructions described above. Each processor must be loaded with the user's program intended for execution. Initially, only one processor is allocated for the execution of the user's program. It serves as the original parent processor, and may or may not spawn child processors to help in its evaluation task. If processors
are available to evaluate code. those portions of code as designated by the parent processor are sent to children processors. Each such processor works independently of the others, except to allocate work, wait for results, or report results. Each processor uses its own data structures and does not share any memory with any other processors.

Of course, the processors must be able to communicate with each other in order to pass initial stacks and results between parent and child. As we have indicated, a message passing system would serve this purpose. Each message would need to be tagged with the id of processor for which it is intended, as well as the id of the sending processor. The sending processor id is required to identify the hole in the parent stack that the result portion of the message should fill. An interface unit in each processor can handle the message managing tasks. Details of such an interface unit are a subject of future work.

4.5 Simulation Enhancements

We have covered many of the additions required in our simulation to handle the cases of finite and infinite numbers of processors. Unlike an implementation of the proposed system, where each processor has its own memory, the simulation had to create the effect of private data structures. Even more importantly, the evaluator had to behave as though the specified number of processors were available, and used as necessary.

Each processor has a set of data structures created for it at the time of allocation. These data structures are managed only by fork, join, and do_top. The fork function takes care of the creation of the processor’s stack and id, while join takes care of deallocation. The control function do_top is greatly enhanced for the multiprocessor system as compared to Jones’ original function. Since we had to simulate many processors, we used a round robin approach in giving actual CPU time to each of the active processors. Every call to do_top forces a context switch to another processor. The effect of this switch is to suspend the current processor and to activate the next processor on the ready list. The
newly activated processor accesses its own data structures. Thus, do_top must also
manage the private data structures. These are kept in a list also, ordered in one-to-one
correspondence with the ready list.

4.6 Examples of Our Multiprocessing Scheme

First, we will solve an example using our sequential method, then investigate how to
use multiprocessing to evaluate the same function.

Example 7

\[
\begin{align*}
\text{let } & x = 1 \\
\text{in} & \\
\text{let } & f (a) = a + x \\
\text{in} & \\
& f (f (2))
\end{align*}
\]

Lambda calculus notation: \[\lambda x . (\lambda f . f (f (2))) (\lambda a . a + x)\]

Abstraction:

\[
\begin{align*}
([x] ([f] ([f] f (f 2)) ([a] + a x)) 1 \\
([x] (S_I^1 ([f] f) ([f] f 2)) (S_I^2 ([a] +) ([a] a) ([a] x))) 1 \\
([x] (S_I^1 I (S_I^1 ([f] f) ([f] 2)) (S_I^2 (K_1 +) I (K_1 x))) 1 \\
([x] (S_I^1 I (S_I^1 I (K_1 2)) (Z_I^{2,1} + I x)) 1 \\
([x] (S_I^1 I (Y_I^{1,1} I 2)) (Z_I^{2,1} + I x)) 1 \\
(S_I^1 ([x] S_I^1 I (Y_I^{1,1} I 2)) ([x] Z_I^{2,1} + I x)) 1 \\
(S_I^1 (S_I^2 ([x] S_I^1)) ([x] I) ([x] Y_I^{1,1} I 2)) \\
(S_I^1 (Z_I^{2,1} ([x] Z_I^{2,1} ([x] +) ([x] I) ([x] x))) 1 \\
(S_I^1 (S_I^2 (K_1 S_I^1)) (K_1 I) (S_I^2 ([x] Y_I^{1,1}) ([x] I) ([x] 2)) \\
(S_I^1 (K_1 Z_I^{2,1}) (K_1 +) (K_1 I) I)) 1 \\
(S_I^1 (S_I^2 (K_1 S_I^1)) I (S_I^2 (K_1 Y_I^{1,1}) (K_1 I) (K_1 2)) \\
(X_I^{1,3} Z_I^{2,1} + I I)) 1
\end{align*}
\]
\[(S_{1}^{2} (S_{1}^{2} (K_{1} S_{1}^{1}) (K_{1} I) (K_{1} (Y_{1}^{1,1} I 2))) (X_{1}^{3,3} Z_{1}^{2,1} + I I)) 1 \]
\[(S_{1}^{1} (K_{1} (S_{1}^{1} I (Y_{1}^{1,1} I 2))) (X_{1}^{3,3} Z_{1}^{2,1} + I I)) 1 \]
\[X_{1}^{1,1} (S_{1}^{1} I (Y_{1}^{1,1} I 2)) (X_{1}^{3,3} Z_{1}^{2,1} + I I) 1 \]

Reduction:
\[\rightarrow S_{1}^{1} I (Y_{1}^{1,1} I 2) (X_{1}^{3,3} Z_{1}^{2,1} + I I) 1 \]
\[\rightarrow (I (X_{1}^{3,3} Z_{1}^{2,1} + I I)) (Y_{1}^{1,1} I 2 (X_{1}^{3,3} Z_{1}^{2,1} + I I)) 1 \]
\[\rightarrow X_{1}^{3,3} Z_{1}^{2,1} + I I 1 (Y_{1}^{1,1} I 2 (X_{1}^{3,3} Z_{1}^{2,1} + I I) 1) \]
\[\rightarrow Z_{1}^{2,1} + I I (I 1) (Y_{1}^{1,1} I 2 (X_{1}^{3,3} Z_{1}^{2,1} + I I) 1) \]
\[\rightarrow + (I (Y_{1}^{1,1} I 2 (X_{1}^{3,3} Z_{1}^{2,1} + I I) 1)) (I 1) \]
\[\rightarrow + (Y_{1}^{1,1} I 2 (X_{1}^{3,3} Z_{1}^{2,1} + I I) 1) 1 \]
\[\rightarrow + (I (X_{1}^{3,3} Z_{1}^{2,1} + I I 1) 2) 1 \]
\[\rightarrow + ((X_{1}^{3,3} Z_{1}^{2,1} + I I 1) 2) 1 \]
\[\rightarrow + (Z_{1}^{2,1} + I (I 1) 2) 1 \]
\[\rightarrow + (+ (I 2) (I 1)) 1 \]
\[\rightarrow + (+ 2 1) 1 \]
\[\rightarrow + (3) 1 \]
\[\rightarrow 4 \]

Combinator Expression: \[X_{1}^{1,1} (S_{1}^{1} I (Y_{1}^{1,1} I 2)) (X_{1}^{3,3} Z_{1}^{2,1} + I I) 1 \]

Generated Code:
\[
(def \text{ex7}
  \\lambda (lambda ()
    (prog ()
      (bind 'example7 'L10)
      (look_up 'example7)
      (first_call 'done)
      done (finish)
      L10 (push 'L30)
      (push 'L50)
      (push 'L70)
      (do_X 1 1 1)
      (do_top)
      L30 (push 1)
      (pexit)
  )
)
The evaluation steps for a single processor system follow:

```
main_program Example_7
    processors=1
    (stack= (L10))
    (ret_stack= (done))
    (do_top)
    (stack= nil)
    (ret_stack= (done))
    (push L30)
    (stack= (L30))
    (ret_stack= (done))
    (push L50)
    (stack= (L50 L30))
    (ret_stack= (done))
    (push L70)
    (stack= (L70 L50 L30))
    (ret_stack= (done))
    (do_X 1 1 1)
    (stack= (L70 L50 L30)))
```
(do_top)
(stack= ((L50 L30)))
(ret_stack= (done))
push L110
(stack= (L110 (L50 L30)))
(ret_stack= (done))
push L150
(stack= (L150 L110 (L50 L30)))
(ret_stack= (done))
do_newS 1 1
(stack= ((L150 (L50 L30)) (L110 (L50 L30))))
(ret_stack= (done))
do_top
(stack= ((L50 L30) (L110 (L50 L30))))
(ret_stack= (done))
do_
(stack= ((L50 L30) (L110 (L50 L30))))
(ret_stack= (done))
do_top
(stack= (L30 (L110 (L50 L30))))
(ret_stack= (done))
push L90
(stack= (L90 L30 (L110 (L50 L30))))
(ret_stack= (done))
push L130
(stack= (L130 L90 L30 (L110 (L50 L30))))
(ret_stack= (done))
push L190
(stack= (L190 L130 L90 L30 (L110 (L50 L30))))
(ret_stack= (done))
push L230
(stack= (L230 L190 L130 L90 L30 (L110 (L50 L30))))
(ret_stack= (done))
do_X 3 1 3
(stack= (L230 L190 L130 L90 L30 (L110 (L50 L30))))
(ret_stack= (done))
do_top
(stack= (L190 L130 (L90 L30) (L110 (L50 L30))))
(ret_stack= (done))
do_Z 2 1 1
(stack= (L190 (L130 (L110 (L50 L30)))) (L90 L30))
(ret_stack= (done))
do_top
(stack= ((L130 (L110 (L50 L30))) (L90 L30)))
(ret_stack= (done))
(old_call L191)
(stack= ((L130 (L110 (L50 L30))) oldcall))
(ret_stack= (L191 ((L90 L30)) done))
do_top
(stack= ((L110 (L50 L30)) oldcall))
(ret_stack= (L191 ((L90 L30)) done))
do_
(stack= ((L110 (L50 L30)) oldcall))
(ret_stack= (L191 ((L90 L30)) done))

(do_top)
(stack= ((L50 L30) oldcall))
(ret_stack= (L191 ((L90 L30)) done))

(push L170)
(stack= (L170 (L50 L30) oldcall))
(ret_stack= (L191 ((L90 L30)) done))

(push L210)
(stack= (L210 L170 (L50 L30) oldcall))
(ret_stack= (L191 ((L90 L30)) done))

(do_Y 1 1 1)
(stack= ((L210 (L50 L30)) L170 oldcall))
(ret_stack= (L191 ((L90 L30)) done))

(do_top)
(stack= ((L50 L30) L170 oldcall))
(ret_stack= (L191 ((L90 L30)) done))

(do_I)
(stack= ((L50 L30) L170 oldcall))
(ret_stack= (L191 ((L90 L30)) done))

(do_top)
(stack= (L30 L170 oldcall))
(ret_stack= (L191 ((L90 L30)) done))

(push L90)
(stack= (L90 L30 L170 oldcall))
(ret_stack= (L191 ((L90 L30)) done))

(push L130)
(stack= (L130 L90 L30 L170 oldcall))
(ret_stack= (L191 ((L90 L30)) done))

(push L190)
(stack= (L190 L130 L90 L30 L170 oldcall))
(ret_stack= (L191 ((L90 L30)) done))

(push L230)
(stack= (L230 L190 L130 L90 L30 L170 oldcall))
(ret_stack= (L191 ((L90 L30)) done))

(do_X 3 1 3)
(stack= (L230 L190 L130 (L90 L30) L170 oldcall))
(ret_stack= (L191 ((L90 L30)) done))

(do_top)
(stack= (L190 L130 (L90 L30) L170 oldcall))
(ret_stack= (L191 ((L90 L30)) done))

(do_Z 2 1 1)
(stack= (L190 (L130 L170) (L90 L30) oldcall))
(ret_stack= (L191 ((L90 L30)) done))

(do_top)
(stack= ((L130 L170) (L90 L30) oldcall))
(ret_stack= (L191 ((L90 L30)) done))

(old_call L191)
(stack= ((L130 L170) oldcall))
(ret_stack= (L191 ((L90 L30) oldcall) L191 ((L90 L30)) done))

(do_top)
(stack= (L170 oldcall))
(ret_stack= (L191 ((L90 L30) oldcall) L191 ((L90 L30)) done))
(do_j)
  (stack= (L170 oldcall))
  (ret_stack= (L191 ((L90 L30) oldcall) L191 ((L90 L30)) done))
(do_top)
  (stack= (oldcall))
  (ret_stack= (L191 ((L90 L30) oldcall) L191 ((L90 L30)) done))
(push 2)
  (stack= (2 oldcall))
  (ret_stack= (L191 ((L90 L30) oldcall) L191 ((L90 L30)) done))
(pexit)
  (stack= (2))
  (ret_stack= (L191 ((L90 L30) oldcall) L191 ((L90 L30)) done))
(old_call2 L192)
  (stack= ((L90 L30) oldcall))
  (ret_stack= (L192 2 (oldcall) L191 ((L90 L30)) done))
(do_top)
  (stack= (L30 oldcall))
  (ret_stack= (L192 2 (oldcall) L191 ((L90 L30)) done))
(do_j)
  (stack= (L30 oldcall))
  (ret_stack= (L192 2 (oldcall) L191 ((L90 L30)) done))
(do_top)
  (stack= (oldcall))
  (ret_stack= (L192 2 (oldcall) L191 ((L90 L30)) done))
(push 1)
  (stack= (1 oldcall))
  (ret_stack= (L192 2 (oldcall) L191 ((L90 L30)) done))
(pexit)
  (stack= (1))
  (ret_stack= (L192 2 (oldcall) L191 ((L90 L30)) done))
(old_do_plus)
  (stack= (3 oldcall))
  (ret_stack= (L191 ((L90 L30)) done))
(pexit)
  (stack= (3))
  (ret_stack= (L191 ((L90 L30)) done))
(old_call2 L192)
  (stack= ((L90 L30) oldcall))
  (ret_stack= (L192 3 nil done))
(do_top)
  (stack= (L30 oldcall))
  (ret_stack= (L192 3 nil done))
(do_j)
  (stack= (L30 oldcall))
  (ret_stack= (L192 3 nil done))
(do_top)
  (stack= (oldcall))
  (ret_stack= (L192 3 nil done))
(push 1)
  (stack= (1 oldcall))
  (ret_stack= (L192 3 nil done))
(pexit)
  (stack= (1))
Our studies have shown us that the combinator functions indicate when portions of code may be allocated to available processors for parallel evaluation. The following execution trace of example 7 allows us to see exactly what code is allocated. The system is set up to simulate an infinite number of processors (128).

main_program Example_7
  processors=128
  (stack= (L10))
  (ret_stack= (done))
  (do_top)
  (stack= nil)
  (ret_stack= (done))
  (push L30)
  (stack= (L30))
  (ret_stack= (done))
  (push L50)
  (stack= (L50 L30))
  (ret_stack= (done))
  (push L70)
  (stack= (L70 L50 L30))
  (ret_stack= (done))
  (do_X 1 1 1 (Proc_2))
  (stack= (L70 Proc_2))
  (ret_stack= (done))
  (do_top)
  (stack= (Proc_2))
  (ret_stack= (done))
  (push L110)
  (stack= (L110 Proc_2))
  (ret_stack= (done))
  (push L150)
  (stack= (spflag (do_newS 1 1) L150 L110 Proc_2))
  (ret_stack= (done))
  (do_top)
  (stack= (spflag (do_newS 1 1) L150 L110 (timeflag (13 4) (spflag (do_Z 2 1 1) L190 L130 1))))
  (ret_stack= (done))
  (do_top)
  (stack= (L150 L110 (spflag (do_Z 2 1 1) L190 L130 1)))
  (ret_stack= (done))
  (wait 10)
  (stack= (L150 L110 (spflag (do_Z 2 1 1) L190 L130 1)))
  (ret_stack= (done))
(do_newS 1 1 (Proc_4))
(stack= ((L150 (spflag (do_Z 2 1 1) L190 L130 1)) Proc_4))
(ret_stack= (done))
(do_top)
(stack= ((spflag (do_Z 2 1 1) L190 L130 1) Proc_4))
(ret_stack= (done))
(do_L)
(stack= ((spflag (do_Z 2 1 1) L190 L130 1) Proc_4))
(ret_stack= (done))
(do_top)
(stack= (spflag (do_Z 2 1 1) L190 L130 1 Proc_4))
(ret_stack= (done))
(do_top)
(stack= (spflag (do_Z 2 1 1) L190 L130 1 Proc_4))
(ret_stack= (done))
(do_top)
(stack= (spflag (do_Z 2 1 1) L190 L130 1 Proc_4))
(ret_stack= (done))
(do_top)
(stack= (spflag (do_Z 2 1 1) L190 L130 1 (timeflag (16 5) 3)))
(ret_stack= (done))
(do_top)
(stack= (L190 L130 1 3))
(ret_stack= (done))
(wait 13)
(stack= (L190 L130 1 3))
(ret_stack= (done))
(do_Z 2 1 1 (Proc_7))
(stack= (L190 (timeflag (4 2) 3) 1))
(ret_stack= (done))
(do_top)
(stack= (L191 (timeflag (4 2) 3) 1))
(ret_stack= (done))
(call L191)
(stack= (L191 (timeflag (4 2) 3) 1))
(ret_stack= (done))
(do_top)
(stack= (L192 (timeflag (4 2) 3) 1))
(ret_stack= (done))
(call2 L192)
(stack= (L192 (timeflag (4 2) 3) 1))
(ret_stack= (done))
(do_top)
(stack= (3 1))
(ret_stack= (done))
(wait 3)
(stack= (3 1))
(ret_stack= (done))
(do_plus)
(stack= (4))
(ret_stack= (done))
(pexit)
Proc_7  
(L130 3)  
(stack= (3))  
(ret_stack= (nil))  
(do)  
(stack= (3))  
(ret_stack= (nil))  
(do_top)  
(3)  

Proc_4  
(L110 (spflag (do Z 2 1 1) L190 L130 1))  
(stack= ((spflag (do Z 2 1 1) L190 L130 1)))  
(ret_stack= (nil))  
push L170  
(stack= (L170 (spflag (do Z 2 1 1) L190 L130 1)))  
(ret_stack= (nil))  
push L210  
(stack= (L210 L170 (spflag (do Z 2 1 1) L190 L130 1)))  
(ret_stack= (nil))  
(do Y 1 1 1 (Proc_5))  
(stack= (((L210 (spflag (do Z 2 1 1) L190 L130 1)) (timeflag (4 2) 2)))  
(ret_stack= (nil))  
(do_top)  
(stack= ((spflag (do Z 2 1 1) L190 L130 1) (timeflag (4 2) 2)))  
(ret_stack= (nil))  
(do)  
(stack= ((spflag (do Z 2 1 1) L190 L130 1) (timeflag (4 2) 2)))  
(ret_stack= (nil))  
(do_top)  
(stack= (L190 L130 1 2))  
(ret_stack= (nil))  
(wait 1)  
(stack= (L190 L130 1 2))  
(ret_stack= (nil))  
(do Z 2 1 1 (Proc_6))  
(stack= (L190 (timeflag (4 2) 2) 1))  
(ret_stack= (nil))  
(do_top)  
(stack= (L191 (timeflag (4 2) 2) 1))  
(ret_stack= (nil))  
call L191  
(stack= (L191 (timeflag (4 2) 2) 1))  
(ret_stack= (nil))  
(do_top)  
(stack= (L192 (timeflag (4 2) 2) 1))  
(ret_stack= (nil))  
call2 L192  
(stack= (L192 (timeflag (4 2) 2) 1))  
(ret_stack= (nil))  
(do_top)  
(stack= (2 1))  
(ret_stack= (nil))  
(wait 3)  
(stack= (2 1))
(ret_stack= (nil))
(do_plus)
(stack= (3))
(ret_stack= (nil))
(pexit)
(3)

Proc_6 (L130 2)
(stack= (2))
(ret_stack= (nil))
(do_1)
(stack= (2))
(ret_stack= (nil))
(do_top)
(2)

Proc_5 (L170)
(stack= nil)
(ret_stack= (nil))
push 2
(stack= (2))
(ret_stack= (nil))
pexit
(2)

Proc_2 (L50 L30)
(stack= (L30))
(ret_stack= (nil))
push L90
(stack= (L90 L30))
(ret_stack= (nil))
push L130
(stack= (L130 L90 L30))
(ret_stack= (nil))
push L190
(stack= (L190 L130 L90 L30))
(ret_stack= (nil))
push L230
(stack= (L230 L190 L130 L90 L30))
(ret_stack= (nil))
(do_X 3 1 3 (Proc_3))
(stack= (L230 L190 L130 L90 L30))
(ret_stack= (nil))
(do_top)
(stack= (spflag (do_Z 2 1 1) L190 L130 (timeflag (6 2) 1)))
(ret_stack= (nil))
(do_top)
(stack= (L190 L130 1))
(ret_stack= (nil))
(wait 5)
(spflag (do_Z 2 1 1) L190 L130 1)

Proc_3 (L90 L30)
Example 8

let \( f(x, y, z) = x + y + z \) and \( a = 1 \)

\[
\text{in } f(a + 2, a + 3, a + 4)
\]

Lambda calculus notation: \([\lambda (f \cdot a) . f (+ a 2) (+ a 3) (+ a 4)]\)

\((\lambda(x\cdot y\cdot z) . + (+ x y) z . 1)\)

Abstraction:

\[
(\{f \cdot a\} f (+ a 2) (+ a 3) (+ a 4)) ([x\cdot y\cdot z] (+ x y) z . 1)
\]

\( (S_2 \cdot I \cdot (S_2 \cdot ([f \cdot a] + ([f \cdot a] a) ([f \cdot a] 2))) \)

\( (S_2 \cdot ([f \cdot a] + ([f \cdot a] a) ([f \cdot a] 3)) \)

\( (S_2 \cdot ([f \cdot a] + ([f \cdot a] a) ([f \cdot a] 4))) \)

\( (S_2 \cdot (K_3 +) (S_2 \cdot ([x\cdot y\cdot z] + ([x\cdot y\cdot z] x) ([x\cdot y\cdot z] y)) I \cdot 1) \)

\( (S_2 \cdot I \cdot (S_2 \cdot (K_2 +) I \cdot 2) (K_2 2)) \)

\( (S_2 \cdot (K_2 +) I \cdot 2) (K_2 3)) \)

\( (S_2 \cdot (K_2 +) I \cdot 2) (K_2 4)) \)

\( (S_2 \cdot (K_3 +) (S_2 \cdot (K_3 +) I \cdot 1) I \cdot 2) I \cdot 3 . 1) \)

\( (S_2 \cdot I \cdot (Z_{3,1} + I \cdot 2) (Z_{3,1} + I \cdot 2) (Z_{3,1} + I \cdot 2) 4)) \)
\[(S_0^2 (K_3 +) (X_1^2 + I_3^1 I_3^2) I_3^3.1)\]
\[(S_0^2 I_4^1 (Z_2^2 + I_3^2 2) (Z_2^1 + I_3^2 3) (Z_2^2 + I_3^2 4))\]
\[(X_3^2 + (X_3^2 + I_3^3 I_3^4) I_3^5.1)\]

Reduction:

\[-(I_4^1 (X_3^2 + (X_3^2 + I_3^3 I_3^4) I_3^5.1)\]
\[ (Z_2^1 + I_3^2 2 (X_3^2 + (X_3^2 + I_3^3 I_3^4) I_3^5.1))\]
\[ (Z_2^1 + I_3^2 3 (X_3^2 + (X_3^2 + I_3^3 I_3^4) I_3^5.1))\]
\[ (Z_2^1 + I_3^2 4 (X_3^2 + (X_3^2 + I_3^3 I_3^4) I_3^5.1))\]

\[-X_3^2 + (X_3^2 + I_3^3 I_3^4) I_3^5.1\]
\[ (Z_2^1 + I_3^2 2 (X_3^2 + (X_3^2 + I_3^3 I_3^4) I_3^5.1))\]
\[ (Z_2^1 + I_3^2 3 (X_3^2 + (X_3^2 + I_3^3 I_3^4) I_3^5.1))\]
\[ (Z_2^1 + I_3^2 4 (X_3^2 + (X_3^2 + I_3^3 I_3^4) I_3^5.1))\]

\[-+(X_3^2 + I_3^3 I_3^4)\]
\[ ((Z_2^1 + I_3^2 2 (X_3^2 + (X_3^2 + I_3^3 I_3^4) I_3^5.1))\]
\[ (Z_2^1 + I_3^2 3 (X_3^2 + (X_3^2 + I_3^3 I_3^4) I_3^5.1))\]
\[ (Z_2^1 + I_3^2 4 (X_3^2 + (X_3^2 + I_3^3 I_3^4) I_3^5.1))\]

\[-+ (I_3^1 (Z_2^1 + I_3^2 2 (X_3^2 + (X_3^2 + I_3^3 I_3^4) I_3^5.1))\]
\[ (Z_2^1 + I_3^2 3 (X_3^2 + (X_3^2 + I_3^3 I_3^4) I_3^5.1))\]
\[ (Z_2^1 + I_3^2 4 (X_3^2 + (X_3^2 + I_3^3 I_3^4) I_3^5.1))\]

\[-+ (I_3^1 (Z_2^1 + I_3^2 2 (X_3^2 + (X_3^2 + I_3^3 I_3^4) I_3^5.1))\]
\[ (Z_2^1 + I_3^2 3 (X_3^2 + (X_3^2 + I_3^3 I_3^4) I_3^5.1))\]
\[ (Z_2^1 + I_3^2 4 (X_3^2 + (X_3^2 + I_3^3 I_3^4) I_3^5.1))\]

\[-+(I_3^1 (Z_2^1 + I_3^2 2 (X_3^2 + (X_3^2 + I_3^3 I_3^4) I_3^5.1))\]
\[ (Z_2^1 + I_3^2 3 (X_3^2 + (X_3^2 + I_3^3 I_3^4) I_3^5.1))\]
\[ (Z_2^1 + I_3^2 4 (X_3^2 + (X_3^2 + I_3^3 I_3^4) I_3^5.1))\]
\[(Z_2^{2.1} + I_2^{2.1} 3 (X_2^{3.1} + (X_3^{2.1} + I_3^{3.1} I_3^{3.1})))\]
\[(Z_2^{2.1} + I_2^{2.1} 4 (X_3^{2.1} + (X_3^{2.1} + I_3^{3.1} I_3^{3.1})))\]
\[\rightarrow + (+ (Z_2^{2.1} + I_2^{2.1} 2 (X_3^{2.1} + (X_3^{2.1} + I_3^{3.1} I_3^{3.1})))\]
\[(Z_2^{2.1} + I_2^{2.1} 3 (X_3^{2.1} + (X_3^{2.1} + I_3^{3.1} I_3^{3.1})))\]
\[(Z_2^{2.1} + I_2^{2.1} 4 (X_3^{2.1} + (X_3^{2.1} + I_3^{3.1} I_3^{3.1})))\]
\[\rightarrow + (+ (+ (Z_2^{2.1} (X_2^{2.1} + (X_2^{2.1} + I_2^{3.1} I_2^{3.1}))) 2)\]
\[(+ (I_2^{2.1} (X_3^{2.1} + (X_3^{2.1} + I_3^{3.1} I_3^{3.1}))) 3))\]
\[(+ (I_2^{2.1} (X_3^{2.1} + (X_3^{2.1} + I_3^{3.1} I_3^{3.1}))) 4)\]
\[\rightarrow + (+ (1 2) (1 3)) (1 4)\]
\[\rightarrow + 3 4 5\]
\[\rightarrow + 7 5\]

12

Combinator Expression: \(S_2^{3.1} I_2^{3.1} (Z_2^{2.1} + I_2^{2.1} 2) (Z_2^{2.1} + I_2^{2.1} 3) (Z_2^{2.1} + I_2^{2.1} 4) (X_2^{3.1} + I_3^{3.1} I_3^{3.1})\)

Generated Code:

```lisp
(defun ex8
 (lambda ()
  (prog ()
    (bind 'example8 'L10)
    (look_up 'example8)
    (first_call 'done)
    done (finish)
    L10 (push 'L30)
    (push 'L50)
    (push 'L70)
    (push 'L110)
    (push 'L170)
    (push 'L270)
    (do_newS 3 2)
    (do_top)
    L30 (push 1)
    (pexit)
    L50 (push 'L90)
    (push 'L150)
    (push 'L230)
    (do_X 2 3 1)
    (do_top)
  ))
)```
L70 (push 'L130)
 (push 'L210)
 (push 'L330)
 (do_Z 2 2 1)
 (do_top)
L110 (push 'L190)
 (push 'L310)
 (push 'L390)
 (do_Z 2 2 1)
 (do_top)
L170 (push 'L290)
 (push 'L370)
 (push 'L430)
 (do_Z 2 2 1)
 (do_top)
L270 (do_newl 1 2)
 (do_top)
L90 (do_newl 3 3)
 (do_top)
L150 (push 'L250)
 (push 'L350)
 (push 'L410)
 (do_X 2 3 1)
 (do_top)
L230 (call1 'L231)
L231 (call2 'L232)
L232 (do_plus)
 (pexit)
L130 (push 4)
 (pexit)
L210 (do_newl 2 2)
 (do_top)
L330 (call1 'L331)
L331 (call2 'L332)
L332 (do_plus)
 (pexit)
L190 (push 3)
 (pexit)
L310 (do_newl 2 2)
 (do_top)
L390 (call1 'L391)
L391 (call2 'L392)
L392 (do_plus)
 (pexit)
L290 (push 2)
 (pexit)
L370 (do_newl 2 2)
 (do_top)
L430 (call1 'L431)
L431 (call2 'L432)
L432 (do_plus)
 (pexit)
L250 (do_newl 2 3)
The evaluation steps for a single processor system illustrate how this program is executed in a sequential manner.

\[
\begin{align*}
\text{main\_program Example} & \quad \text{Example 8} \\
\text{processors} & = 1 \\
\text{(stack} & = (L10)) \\
\text{(ret\_stack} & = (\text{done})) \\
\text{(do\_top)} \\
\text{(stack} & = \text{nil}) \\
\text{(ret\_stack} & = (\text{done})) \\
\text{(push L30)} \\
\text{(stack} & = (L30)) \\
\text{(ret\_stack} & = (\text{done})) \\
\text{(push L50)} \\
\text{(stack} & = (L50 L30)) \\
\text{(ret\_stack} & = (\text{done})) \\
\text{(push L70)} \\
\text{(stack} & = (L70 L50 L30)) \\
\text{(ret\_stack} & = (\text{done})) \\
\text{(push L110)} \\
\text{(stack} & = (L110 L70 L50 L30)) \\
\text{(ret\_stack} & = (\text{done})) \\
\text{(push L170)} \\
\text{(stack} & = (L170 L110 L70 L50 L30)) \\
\text{(ret\_stack} & = (\text{done})) \\
\text{(push L270)} \\
\text{(stack} & = (L270 L170 L110 L70 L50 L30)) \\
\text{(ret\_stack} & = (\text{done})) \\
\text{(do\_new I 3 2)} \\
\text{(stack} & = (((L270 L50 L30) (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)))) \\
\text{(ret\_stack} & = (\text{done})) \\
\text{(do\_top)} \\
\text{(stack} & = (L50 L30 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30))) \\
\text{(ret\_stack} & = (\text{done})) \\
\text{(do\_new I 1 2)} \\
\text{(stack} & = (L50 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30))) \\
\text{(ret\_stack} & = (\text{done})) \\
\text{(do\_top)} \\
\text{(stack} & = (((L110 L50 L30) (L170 L50 L30) (L70 L50 L30)) \\
\text{(ret\_stack} & = (\text{done})) \\
\text{(push L90)} \\
\text{(stack} & = (L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)))
\end{align*}
\]
(ret_stack= (done))
(push L150)
(stack= (L150 L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)))
(ret_stack= (done))
(push L230)
(stack= (L230 L150 L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)))
(ret_stack= (done))
(do_X 2 3 1)
(stack= (L230 (L150 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30))
(L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30))))
(ret_stack= (done))
(do_top)
(stack= ((L150 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30))
(L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)))
(ret_stack= (done))
(old_call L231)
(stack= ((L150 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) oldcall))
(ret_stack= (L231 ((L90 (L170 L50 L30) (L110 L50 L30)
(L70 L50 L30))) done))
(do_top)
(stack= ((L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) oldcall))
(ret_stack= (L231 ((L90 (L170 L50 L30) (L110 L50 L30)
(L70 L50 L30))) done))
(push L250)
(stack= (L250 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30) oldcall))
(ret_stack= (L231 ((L90 (L170 L50 L30) (L110 L50 L30)
(L70 L50 L30))) done))
(push L350)
(stack= (L350 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30) oldcall))
(ret_stack= (L231 ((L90 (L170 L50 L30) (L110 L50 L30)
(L70 L50 L30))) done))
(push L410)
(stack= (L410 (L350 (L170 L50 L30) (L110 L50 L30)
(L70 L50 L30)) oldcall))
(ret_stack= (L231 ((L90 (L170 L50 L30) (L110 L50 L30)
(L70 L50 L30))) done))
(do_X 2 3 1)
(stack= (L410 (L350 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30))
(L250 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) oldcall))
(ret_stack= (L231 ((L90 (L170 L50 L30) (L110 L50 L30)
(L70 L50 L30))) done))
(old_call L411)
(stack= ((L350 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) oldcall))
(ret_stack= (L411 ((L250 (L170 L50 L30) (L110 L50 L30)
(L70 L50 L30)) oldcall) L231 ((L90 (L170 L50 L30)
(L110 L50 L30) (L70 L50 L30)) done))
(do_top)
(stack= ((L170 L50 L30) (L110 L50 L30) (L70 L50 L30) oldcall)))
(ret_stack= (L411 ((L250 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) oldcall) L231 ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30))) done))

(do_newI 1 3)
(stack= ((L170 L50 L30) oldcall))
(ret_stack= (L411 ((L250 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) oldcall) L231 ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30))) done))

(do_top)
(stack= (L50 L30 oldcall))
(ret_stack= (L411 ((L250 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) oldcall) L231 ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30))) done))

(push L290)
(stack= (L290 L50 L30 oldcall))
(ret_stack= (L411 ((L250 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) oldcall) L231 ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30))) done))

(push L370)
(stack= (L370 L290 L50 L30 oldcall))
(ret_stack= (L411 ((L250 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) oldcall) L231 ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30))) done))

(push L430)
(stack= (L430 L370 L290 L50 L30 oldcall))
(ret_stack= (L411 ((L250 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) oldcall) L231 ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30))) done))

(do_Z 2 2 1)
(stack= (L430 (L370 L50 L30) L290 oldcall))
(ret_stack= (L411 ((L250 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) oldcall) L231 ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30))) done))

(do_top)
(stack= ((L370 L50 L30) L290 oldcall))
(ret_stack= (L411 ((L250 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) oldcall) L231 ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30))) done))

(old_call1 L431)
(stack= ((L170 L50 L30) oldcall))
(ret_stack= (L431 (L290 oldcall) L411 ((L250 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) oldcall) L231 ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30))) done))

(do_top)
(stack= (L50 L30 oldcall))
(ret_stack= (L431 (L290 oldcall) L411 ((L250 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) oldcall) L231 ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30))) done))

(do_newI 2 2)
(stack= (L30 oldcall))
(ret_stack= (L431 (L290 oldcall) L411 ((L250 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) oldcall) L231 ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30))) done))
(do_top)
(stack= (oldcall))
(ret_stack= (L431 (L290 oldcall) L411 ((L250 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) oldcall) L231 ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) done))

(push 1)
(stack= (1 oldcall))
(ret_stack= (L431 (L290 oldcall) L411 ((L250 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) oldcall) L231 ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) done))

(pexit)
(stack= (1))
(ret_stack= (L431 (L290 oldcall) L411 ((L250 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) oldcall) L231 ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) done))

(old_call2 L432)
(stack= (L290 oldcall))
(ret_stack= (L432 1 (oldcall) L411 ((L250 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) oldcall) L231 ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) done))

(do_top)
(stack= (oldcall))
(ret_stack= (L432 1 (oldcall) L411 ((L250 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) oldcall) L231 ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) done))

(push 2)
(stack= (2 oldcall))
(ret_stack= (L432 1 (oldcall) L411 ((L250 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) oldcall) L231 ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) done))

(pexit)
(stack= (2))
(ret_stack= (L432 1 (oldcall) L411 ((L250 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) oldcall) L231 ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) done))

(old_do_plus)
(stack= (3 oldcall))
(ret_stack= (L411 ((L250 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) oldcall) L231 ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) done))

(pexit)
(stack= (3))
(ret_stack= (L411 ((L250 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) oldcall) L231 ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) done))

(old_call2 L412)
(stack= ((L250 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) oldcall))
(ret_stack= (L412 3 (oldcall) L231 ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) done))

(do_top)
(stack= ((L170 L50 L30) (L110 L50 L30) (L70 L50 L30) oldcall))
(ret_stack= (L412 3 (oldcall) L231 ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30)) done))
(do_newl 2 3)
  (stack= ((L110 L50 L30) oldcall))
  (ret_stack= (L412 3 (oldcall) L231 ((L90 (L170 L50 L30)
    (L110 L50 L30) (L70 L50 L30))) done))

(do_top)
  (stack= (L50 L30 oldcall))
  (ret_stack= (L412 3 (oldcall) L231 ((L90 (L170 L50 L30)
    (L110 L50 L30) (L70 L50 L30))) done))

(push L190)
  (stack= (L190 L50 L30 oldcall))
  (ret_stack= (L412 3 (oldcall) L231 ((L90 (L170 L50 L30)
    (L110 L50 L30) (L70 L50 L30))) done))

(push L310)
  (stack= (L310 L190 L50 L30 oldcall))
  (ret_stack= (L412 3 (oldcall) L231 ((L90 (L170 L50 L30)
    (L110 L50 L30) (L70 L50 L30))) done))

(push L390)
  (stack= (L390 L310 L190 L50 L30 oldcall))
  (ret_stack= (L412 3 (oldcall) L231 ((L90 (L170 L50 L30)
    (L110 L50 L30) (L70 L50 L30))) done))

(do_Z 2 2 1)
  (stack= (L390 (L310 L50 L30) L190 oldcall))
  (ret_stack= (L412 3 (oldcall) L231 ((L90 (L170 L50 L30)
    (L110 L50 L30) (L70 L50 L30))) done))

(do_top)
  (stack= ((L310 L50 L30) L190 oldcall))
  (ret_stack= (L412 3 (oldcall) L231 ((L90 (L170 L50 L30)
    (L110 L50 L30) (L70 L50 L30))) done))

(old_call1 L391)
  (stack= ((L310 L50 L30) oldcall))
  (ret_stack= (L391 (L190 oldcall) L412 3 (oldcall) L231
    ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30))) done))

(do_top)
  (stack= (L50 L30 oldcall))
  (ret_stack= (L391 (L190 oldcall) L412 3 (oldcall) L231
    ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30))) done))

(do_newl 2 2)
  (stack= (L30 oldcall))
  (ret_stack= (L391 (L190 oldcall) L412 3 (oldcall) L231
    ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30))) done))

(do_top)
  (stack= (oldcall))
  (ret_stack= (L391 (L190 oldcall) L412 3 (oldcall) L231
    ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30))) done))

(push 1)
  (stack= (1 oldcall))
  (ret_stack= (L391 (L190 oldcall) L412 3 (oldcall) L231
    ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30))) done))

(pexit)
  (stack= (1))
  (ret_stack= (L391 (L190 oldcall) L412 3 (oldcall) L231
    ((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30))) done))

(old_call2 L392)
(stack= (L190 oldcall))
(ret_stack= (L392 1 (oldcall) L412 3 (oldcall) L231
((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30))) done))

do_top
(stack= (oldcall))
(ret_stack= (L392 1 (oldcall) L412 3 (oldcall) L231
((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30))) done))

(push 3)
(stack= (3 oldcall))
(ret_stack= (L392 1 (oldcall) L412 3 (oldcall) L231
((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30))) done))

(pexit)
(stack= (3))
(ret_stack= (L392 1 (oldcall) L412 3 (oldcall) L231
((L90 (L170 L50 L30) (L110 L50 L30) (L70 L50 L30))) done))

(ol_d_do_plus)
(stack= (4 oldcall))
(ret_stack= (L412 3 (oldcall) L231 ((L90 (L170 L50 L30)
(L110 L50 L30) (L70 L50 L30))) done))

(pexit)
(stack= (4))
(ret_stack= (L412 3 (oldcall) L231 ((L90 (L170 L50 L30)
(L110 L50 L30) (L70 L50 L30))) done))

(ol_d_do_plus)
(stack= (7 oldcall))
(ret_stack= (L231 ((L90 (L170 L50 L30) (L110 L50 L30)
(L70 L50 L30))) done))

(pexit)
(stack= (7))
(ret_stack= (L231 ((L90 (L170 L50 L30) (L110 L50 L30)
(L70 L50 L30))) done))

(old_call2 L232)
(stack= (L232 7 nil done)
(ret_stack= (L232 7 nil done))

do_top
(stack= ((L170 L50 L30) (L110 L50 L30) (L70 L50 L30) oldcall))
(ret_stack= (L232 7 nil done))

do_newI 3 3
(stack= ((L70 L50 L30) oldcall))
(ret_stack= (L232 7 nil done))

do_top
(stack= (L50 L30 oldcall))
(ret_stack= (L232 7 nil done))

(push L130)
(stack= (L130 L50 L30 oldcall))
(ret_stack= (L232 7 nil done))

(push L210)
(stack= (L210 L130 L50 L30 oldcall))
(ret_stack= (L232 7 nil done))

(push L330)
(stack= (L330 L210 L130 L50 L30 oldcall))
(ret_stack= (L232 7 nil done))

(do_Z 2 2 1)
The evaluation of example 8 is next illustrated using a system consisting of an infinite (128) number of processors. This system setup will allow for the maximum amount of parallelism that our combinators support to occur.
main_program  Example_8
  (stack= (L10))
  (ret_stack= (done))
  (do_top)
   (stack= nil)
   (ret_stack= (done))
  (push L30)
   (stack= (L30))
   (ret_stack= (done))
  (push L50)
   (stack= (L50 L30))
   (ret_stack= (done))
  (push L70)
   (stack= (L70 L50 L30))
   (ret_stack= (done))
  (push L110)
   (stack= (L110 L70 L50 L30))
   (ret_stack= (done))
  (push L170)
   (stack= (L170 L110 L70 L50 L30))
   (ret_stack= (done))
  (push L270)
   (stack= (L270 L170 L110 L70 L50 L30))
   (ret_stack= (done))
  (do_new S 3 2 (Proc_2) (Proc_3) (Proc_4))
   (stack= ((L270 L50 L30) Proc_2 Proc_3 Proc_4))
   (ret_stack= (done))
  (do_top)
   (stack= (L50 L30 Proc_2 Proc_3 Proc_4))
   (ret_stack= (done))
  (do_new I 1 2)
   (stack= (L50 Proc_2 Proc_3 Proc_4))
   (ret_stack= (done))
  (do_top)
   (stack= (Proc_2 Proc_3 Proc_4))
   (ret_stack= (done))
  (push L90)
   (stack= (L90 Proc_2 Proc_3 Proc_4))
   (ret_stack= (done))
  (push L150)
   (stack= (L150 L90 Proc_2 Proc_3 Proc_4))
   (ret_stack= (done))
  (push L230)
   (stack= (spflag (do_X 2 3 1) L230 L150 L90 Proc_2 Proc_3 Proc_4))
   (ret_stack= (done))
  (do_top)
   (stack= (spflag (do_X 2 3 1) L230 L150 L90 (timeflag (14 4) 3)
    (timeflag (14 4) 4) (timeflag (14 4) 5)))
   (ret_stack= (done))
  (do_top)
   (stack= (L230 L150 L90 3 4 5))
   (ret_stack= (done))
(wait 8)
(stack= (L230 L150 L90 3 4 5))
(ret_stack= (done))
(do_X 2 3 1 (Proc_11) (Proc_12))
(stack= (L230 Proc_11 (timeflag (4 2) 5)))
(ret_stack= (done))
(do_top)
(stack= (L231 Proc_11 (timeflag (4 2) 5)))
(ret_stack= (done))
(call1 L231)
(stack= (L231 Proc_11 (timeflag (4 2) 5)))
(ret_stack= (done))
(do_top)
(stack= (L232 Proc_11 (timeflag (4 2) 5)))
(ret_stack= (done))
(call2 L232)
(stack= (L232 Proc_11 (timeflag (4 2) 5)))
(ret_stack= (done))
(do_top)
(stack= ((timeflag (12 4) 7) 5))
(ret_stack= (done))
(wait 3)
(stack= (L232 (timeflag (12 4) 7) 5))
(ret_stack= (done))
(do_top)
(stack= (7 5))
(ret_stack= (done))
(wait 12)
(stack= (7 5))
(ret_stack= (done))
(do_plus)
(stack= (12))
(ret_stack= (done))
(pexit)

Proc_11  (L150 3 4 5)
(stack= (3 4 5))
(ret_stack= (nil))
(push L250)
(stack= (L250 3 4 5))
(ret_stack= (nil))
(push L350)
(stack= (L350 L250 3 4 5))
(ret_stack= (nil))
(push L410)
(stack= (L410 L350 L250 3 4 5))
(ret_stack= (nil))
(do_X 2 3 1 (Proc_13) (Proc_14))
(stack= (L410 (timeflag (4 2) 3) (timeflag (4 2) 4)))
(ret_stack= (nil))
(do_top)
(stack= (L411 (timeflag (4 2) 3) (timeflag (4 2) 4)))
(ret_stack= (nil))
(call1 L411)
  (stack= (L411 (timeflag (4 2) 3) (timeflag (4 2) 4)))
  (ret_stack= (nil))
(do_top)
  (stack= (L412 (timeflag (4 2) 3) (timeflag (4 2) 4)))
  (ret_stack= (nil))
(call2 L412)
  (stack= (L412 (timeflag (4 2) 3) (timeflag (4 2) 4)))
  (ret_stack= (nil))
(do_top)
  (stack= (3 4))
  (ret_stack= (nil))
(wait 3)
  (stack= (3 4))
  (ret_stack= (nil))
(do_plus)
  (stack= (7))
  (ret_stack= (nil))
(pexit)
(7)
Proc_14 (L250 3 4 5)
  (stack= (3 4 5))
  (ret_stack= (nil))
(do_newl 2 3)
  (stack= (4))
  (ret_stack= (nil))
(do_top)
(4)
Proc_13 (L350 3 4 5)
  (stack= (3 4 5))
  (ret_stack= (nil))
(do_newl 1 3)
  (stack= (3))
  (ret_stack= (nil))
(do_top)
(3)
Proc_12 (L90 3 4 5)
  (stack= (3 4 5))
  (ret_stack= (nil))
(do_newl 1 3)
  (stack= (5))
  (ret_stack= (nil))
(do_top)
(5)
Proc_4 (L70 L50 L30)
  (stack= (1.50 L30))
  (ret_stack= (nil))
(push L130)
  (stack= (1.130 L50 L30))
(ret_stack= (nil))
(push L210)
  (stack= (L210 L130 L50 L30))
  (ret_stack= (nil))
(push L330)
  (stack= (L330 L210 L130 L50 L30))
  (ret_stack= (nil))
(do Z 2 2 1 (Proc_9) (Proc_10))
  (stack= (L330 (timeflag (6 2) 1) (timeflag (4 2) 4)))
  (ret_stack= (nil))
(do_top)
  (stack= (L331 (timeflag (6 2) 1) (timeflag (4 2) 4)))
  (ret_stack= (nil))
(call1 L331)
  (stack= (L331 (timeflag (6 2) 1) (timeflag (4 2) 4)))
  (ret_stack= (nil))
(do_top)
  (stack= (L332 (timeflag (6 2) 1) (timeflag (4 2) 4)))
  (ret_stack= (nil))
(call2 L332)
  (stack= (L332 (timeflag (6 2) 1) (timeflag (4 2) 4)))
  (ret_stack= (nil))
(do_top)
  (stack= (1 4))
  (ret_stack= (nil))
(wait 5)
  (stack= (1 4))
  (ret_stack= (nil))
(do_plus)
  (stack= (5))
  (ret_stack= (nil))
(pexit)
(5)

Proc_3 (L110 L50 L30)
  (stack= (L50 L30))
  (ret_stack= (nil))
(push L190)
  (stack= (L190 L50 L30))
  (ret_stack= (nil))
(push L310)
  (stack= (L310 L190 L50 L30))
  (ret_stack= (nil))
(push L390)
  (stack= (L390 L310 L190 L50 L30))
  (ret_stack= (nil))
(do Z 2 2 1 (Proc_7) (Proc_8))
  (stack= (L390 (timeflag (6 2) 1) (timeflag (4 2) 3)))
  (ret_stack= (nil))
(do_top)
  (stack= (L391 (timeflag (6 2) 1) (timeflag (4 2) 3)))
  (ret_stack= (nil))
(call1 L391)
(stack= (L391 (timeflag (6 2) 1) (timeflag (4 2) 3)))
(ret_stack= (nil))
(do_top)
(stack= (L392 (timeflag (6 2) 1) (timeflag (4 2) 3)))
(ret_stack= (nil))
(call2 L392)
(stack= (L392 (timeflag (6 2) 1) (timeflag (4 2) 3)))
(ret_stack= (nil))
(do_top)
(stack= (1 3))
(ret_stack= (nil))
(wait 5)
(stack= (1 3))
(ret_stack= (nil))
(do_plus)
(stack= (4))
(ret_stack= (nil))
(pexit)
(4)

Proc_2 (L170 L50 L30)
(stack= (L50 L30))
(ret_stack= (nil))
(push L290)
(stack= (L290 L50 L30))
(ret_stack= (nil))
(push L370)
(stack= (L370 L290 L50 L30))
(ret_stack= (nil))
(push L430)
(stack= (L430 L370 L290 L50 L30))
(ret_stack= (nil))
(do_2 2 1 (Proc_5) (Proc_6))
(stack= (L430 (timeflag (6 2) 1) (timeflag (4 2) 2)))
(ret_stack= (nil))
(do_top)
(stack= (L431 (timeflag (6 2) 1) (timeflag (4 2) 2)))
(ret_stack= (nil))
(call1 L431)
(stack= (L431 (timeflag (6 2) 1) (timeflag (4 2) 2)))
(ret_stack= (nil))
(do_top)
(stack= (L432 (timeflag (6 2) 1) (timeflag (4 2) 2)))
(ret_stack= (nil))
(call2 L432)
(stack= (L432 (timeflag (6 2) 1) (timeflag (4 2) 2)))
(ret_stack= (nil))
(do_top)
(stack= (1 2))
(ret_stack= (nil))
(wait 5)
(stack= (1 2))
(ret_stack= (nil))
(do_plus)
  (stack= (3))
  (ret_stack= (nil))
(pexit)
(3)

Proc_9 (L210 L50 L30)
  (stack= (L50 L30))
  (ret_stack= (nil))
(do_newl 2 2)
  (stack= (L30))
  (ret_stack= (nil))
(do_top)
  (stack= nil)
  (ret_stack= (nil))
(push 1)
  (stack= (1))
  (ret_stack= (nil))
(pexit)
(1)

Proc_7 (L310 L50 L30)
  (stack= (L50 L30))
  (ret_stack= (nil))
(do_newl 2 2)
  (stack= (L30))
  (ret_stack= (nil))
(do_top)
  (stack= nil)
  (ret_stack= (nil))
(push 1)
  (stack= (1))
  (ret_stack= (nil))
(pexit)
(1)

Proc_5 (L370 L50 L30)
  (stack= (L50 L30))
  (ret_stack= (nil))
(do_newl 2 2)
  (stack= (L30))
  (ret_stack= (nil))
(do_top)
  (stack= nil)
  (ret_stack= (nil))
(push 1)
  (stack= (1))
  (ret_stack= (nil))
(pexit)
(1)

Proc_10 (L130)
  (stack= nil)
To further illustrate the capability of our system to evaluate combinator expressions in parallel, we will revisit some of our previous examples. Evaluation of these examples will be conducted with the system set up with an infinite (128) number of processors.

Example 3

\[
\begin{align*}
\text{let } a &= 1 \\
\text{in} \\
\text{let } b &= 2 \\
\text{in} \\
a + b
\end{align*}
\]

Combinator Expression: $Y \, 1 \downarrow \downarrow (Z \, i^2 \downarrow \downarrow X \, i^2 \downarrow \downarrow + \downarrow \downarrow I \, I) \, 2 \, 1$

Generated Code: (repeated here for convenience)

\[
\begin{align*}
(def \, ex3
\quad (\lambda ()
\quad \begin{align*}
\text{(bind 'example3 'L10) }
\text{(first_call 'done) }
\text{done (finish) } \\
&L10 \, \text{(push 'L30)}
\end{align*}
\end{align*}
\]

\]
(push 'L50)
(push 'L70)
(do_Y 1 1 1)
(do_top)
L30  (push 1)
  (pexit)
L50  (push 2)
  (pexit)
L70  (push 'L90)
  (push 'L110)
  (push 'L130)
  (push 'L150)
  (do_Z 3 1 2)
  (do_top)
L90  (do_I)
  (do_top)
L110 (do_I)
  (do_top)
L130  (call1 'L131)
L131  (call2 'L132)
L132  (do_plus)
  (pexit)
L150  (do_X 2 1 2)
  (do_top)))

Evaluation steps:

main_program Example_3
  processors=128
  (stack= (L10))
  (ret_stack= (done))
  (do_top)
    (stack= nil)
    (ret_stack= (done))
  (push L30)
    (stack= (L30))
    (ret_stack= (done))
  (push L50)
    (stack= (L50 L30))
    (ret_stack= (done))
  (push L70)
    (stack= (L70 L50 L30))
    (ret_stack= (done))
  (do_Y 1 1 1 (Proc_2))
    (stack= ((L70 L30) (timeflag (4 2) 2)))
    (ret_stack= (done))
  (do_top)
    (stack= (L30 (timeflag (4 2) 2)))
    (ret_stack= (done))
  (push L90)
    (stack= (L90 L30 (timeflag (4 2) 2)))
    (ret_stack= (done))
(push L110)
  (stack= (L110 L90 L30 (timeflag (4 2) 2)))
  (ret_stack= (done))
(push L130)
  (stack= (L130 L110 L90 L30 (timeflag (4 2) 2)))
  (ret_stack= (done))
(push L150)
  (stack= (L150 L130 L110 L90 L30 (timeflag (4 2) 2)))
  (ret_stack= (done))
(do Z 3 1 2 (Proc_3) (Proc_4))
  (stack= (L150 L130 Proc_3 Proc_4 (timeflag (4 2) 2)))
  (ret_stack= (done))
(do_top)
  (stack= (L130 (timeflag (6 2) 1) (timeflag (4 2) nil) 2))
  (ret_stack= (done))
(wait 3)
  (stack= (spflag (do_X 2 1 2) L130 (timeflag (6 2) 1) (timeflag (4 2) nil) 2))
  (ret_stack= (done))
(do_top)
  (stack= (L130 1 nil 2))
  (ret_stack= (done))
(wait 2)
  (stack= (L130 1 nil 2))
  (ret_stack= (done))
(do_X 2 1 2 (Proc_5))
  (stack= (L130 1 (timeflag (2 2) 2)))
  (ret_stack= (done))
(do_top)
  (stack= (L131 1 (timeflag (2 2) 2)))
  (ret_stack= (done))
(call L131)
  (stack= (L131 1 (timeflag (2 2) 2)))
  (ret_stack= (done))
(do_top)
  (stack= (L132 1 (timeflag (2 2) 2)))
  (ret_stack= (done))
(call2 L132)
  (stack= (L132 1 (timeflag (2 2) 2)))
  (ret_stack= (done))
(do_top)
  (stack= (1 2))
  (ret_stack= (done))
(wait 1)
  (stack= (1 2))
  (ret_stack= (done))
(do_plus)
  (stack= (3))
  (ret_stack= (done))
pexit

Proc_5 (nil 2)
(2)
Proc_4 (L90)
(stack= (nil))
(ret_stack= (nil))
(do_1)
(stack= (nil))
(ret_stack= (nil))
(do_top)
(nil)

Proc_3 (L110 L30)
(stack= (L30))
(ret_stack= (nil))
(do_1)
(stack= (L30))
(ret_stack= (nil))
(do_top)
(stack= nil)
(ret_stack= (nil))
(push 1)
(stack= (1))
(ret_stack= (nil))
(pexit)
(1)

Proc_2 (L50)
(stack= nil)
(ret_stack= (nil))
(push 2)
(stack= (2))
(ret_stack= (nil))
(pexit)
(2)

Example 4

let a = 1 and b = 2
in
  a + b

Combinator Expression: + 1 2

Generated Code:

(def ex4
  (lambda ()
    (prog ()
      (bind 'example4 'L10)
      (look_up 'example4)
      (first_call 'done)
      (finish)
      (push 'L10)
      (push 'L50)
      (push 'L30)
(call1 'L41)
L41 (call2 'L42)
L42 (do_plus)
(pexit)
L30 (push 2)
(pexit)
L50 (push 1)
(pexit))))

Evaluation steps:

main_program Example_4
  processors=128
  (stack= (L10))
  (ret_stack= (done))
  (do_top)
  (stack= nil)
  (ret_stack= (done))
  (push L30)
  (stack= (L30))
  (ret_stack= (done))
  (push L50)
  (stack= (L41 L50 L30))
  (ret_stack= (done))
  (call1 L41)
  (stack= (L41 L50 L30))
  (ret_stack= (done))
  (do_top)
  (stack= (L42 L50 L30))
  (ret_stack= (done))
  (call2 L42)
  (stack= (L42 L50 L30))
  (ret_stack= (done))
  (do_top (Proc_2) (Proc_3))
  (stack= (L42 (timeflag (4 2) 1) (timeflag (4 2) 2)))
  (ret_stack= (done))
  (do_top)
  (stack= (1 2))
  (ret_stack= (done))
  (wait 4)
  (stack= (1 2))
  (ret_stack= (done))
  (do_plus)
  (stack= (3))
  (ret_stack= (done))
  (pexit)

Proc_3 (L30)
  (stack= nil)
  (ret_stack= (nil))
  (push 2)
  (stack= (2))
Example 5

\[
\text{let } a = 1 \text{ and } b = 2 \text{ and } c = 3 \\
\text{in } \hspace{1cm} a + b + c
\]

Combinator Expression: \( X_3^{2,1} + (X_3^{2,1} + I_3^1 I_3^2) I_3^3 1 2 3 \)

Generated Code:

```lisp
(def ex5
  (lambda ()
    (prog ()
      (bind 'example5 'L10)
      (look_up 'example5)
      (first_call 'done)
      done (finish)
    L10 (push 'L30)
      (push 'L50)
      (push 'L70)
      (push 'L90)
      (push 'L110)
      (push 'L130)
      (do X 2 3 1)
      (do_top)
    L30 (push 3)
      (pexit)
    L50 (push 2)
      (pexit)
    L70 (push 1)
      (pexit)
    L90 (do_newl 3 3)
      (do_top)
    L110 (push 'L150)
      (push 'L170)
      (push 'L190)
      (do X 2 3 1)
      (do_top)
    L130 (calli 'L131)
```

L131 (call2 'L132)
L132 (do_plus)
  (pexit)
L150 (do_newI 2 3)
  (do_top)
L170 (do_newI 1 3)
  (do_top)
L190 (call1 'L191)
L191 (call2 'L192)
L192 (do_plus)
  (pexit)))

Evaluation Steps:

main_program  Example_5
  processors=128
  (stack= (L10))
  (ret_stack= (done))
  (do_top)
    (stack= nil)
    (ret_stack= (done))
  (push L30)
    (stack= (L30))
    (ret_stack= (done))
  (push L50)
    (stack= (L50 L30))
    (ret_stack= (done))
  (push L70)
    (stack= (L70 L50 L30))
    (ret_stack= (done))
  (push L90)
    (stack= (L90 L70 L50 L30))
    (ret_stack= (done))
  (push L110)
    (stack= (L110 L90 L70 L50 L30))
    (ret_stack= (done))
  (push L130)
    (stack= (L130 L110 L90 L70 L50 L30))
    (ret_stack= (done))
  (do_X 2 3 1 (Proc_2) (Proc_3))
    (stack= (L130 Proc_2 (timeflag (6 2) 3)))
    (ret_stack= (done))
  (do_top)
    (stack= (L131 Proc_2 (timeflag (6 2) 3)))
    (ret_stack= (done))
  (call1 L131)
    (stack= (L131 Proc_2 (timeflag (6 2) 3)))
    (ret_stack= (done))
  (do_top)
    (stack= (L132 Proc_2 (timeflag (6 2) 3)))
    (ret_stack= (done))
(call2 L132)
  (stack= (L132 Proc_2 (timeflag (6 2) 3)))
  (ret_stack= (done))
(do_top)
  (stack= ((timeflag (14 4) 3) 3))
  (ret_stack= (done))
(wait 5)
  (stack= (L132 (timeflag (14 4) 3) 3))
  (ret_stack= (done))
(do_top)
  (stack= (3 3))
  (ret_stack= (done))
(wait 14)
  (stack= (3 3))
  (ret_stack= (done))
(do_plus)
  (stack= (6))
  (ret_stack= (done))
(pexit)

Proc_2
  (L110 L70 L50 L30)
  (stack= (L70 L50 L30))
  (ret_stack= (nil))
(push L150)
  (stack= (L150 L70 L50 L30))
  (ret_stack= (nil))
(push L170)
  (stack= (L170 L150 L70 L50 L30))
  (ret_stack= (nil))
(push L190)
  (stack= (L190 L170 L150 L70 L50 L30))
  (ret_stack= (nil))
(do_X 2 3 1 (Proc_4) (Proc_5))
  (stack= (L190 (timeflag (6 2) 1) (timeflag (6 2) 2)))
  (ret_stack= (nil))
(do_top)
  (stack= (L191 (timeflag (6 2) 1) (timeflag (6 2) 2)))
  (ret_stack= (nil))
(call1 L191)
  (stack= (L191 (timeflag (6 2) 1) (timeflag (6 2) 2)))
  (ret_stack= (nil))
(do_top)
  (stack= (L192 (timeflag (6 2) 1) (timeflag (6 2) 2)))
  (ret_stack= (nil))
(call2 L192)
  (stack= (L192 (timeflag (6 2) 1) (timeflag (6 2) 2)))
  (ret_stack= (nil))
(do_top)
  (stack= (1 2))
  (ret_stack= (nil))
(wait 5)
  (stack= (1 2))
  (ret_stack= (nil))
(do_plus)
  (stack= (3))
  (ret_stack= (nil))
(pexit)
(3)

Proc_5 (L150 L70 L50 L30)
  (stack= (L70 L50 L30))
  (ret_stack= (nil))
(do_newI 2 3)
  (stack= (L50))
  (ret_stack= (nil))
(do_top)
  (stack= nil)
  (ret_stack= (nil))
(push 2)
  (stack= (2))
  (ret_stack= (nil))
(pexit)
(2)

Proc_4 (L170 L70 L50 L30)
  (stack= (L70 L50 L30))
  (ret_stack= (nil))
(do_newI 1 3)
  (stack= (L70))
  (ret_stack= (nil))
(do_top)
  (stack= nil)
  (ret_stack= (nil))
(push 1)
  (stack= (1))
  (ret_stack= (nil))
(pexit)
(1)

Proc_3 (L90 L70 L50 L30)
  (stack= (L70 L50 L30))
  (ret_stack= (nil))
(do_newI 3 3)
  (stack= (L30))
  (ret_stack= (nil))
(do_top)
  (stack= nil)
  (ret_stack= (nil))
(push 3)
  (stack= (3))
  (ret_stack= (nil))
(pexit)
(3)
Example 6

\[
\text{let } f \ (a,b) = a + b \\
\text{in } f \ (f \ (3.4),1)
\]

Combinator Expression: \( Y^{2.4} I \ (Z^{2.9} I \ 3 \ 4) \ 1 + \)

Generated Code:

```lisp
(def ex6
  (lambda ()
    (prog ()
      (bind 'example6 'L10)
      (look_up 'example6)
      (first_call 'done)
      done (finish)
      L10 (push 'L30)
      (push 'L50)
      (push 'L70)
      (push 'L90)
      (do_Y 2 1 2)
      (do_top)
      L30 (call1 'L31)
      L31 (call2 'L32)
      L32 (do_plus)
      (pexit)
      L50 (push 1)
      (pexit)
      L70 (push 'L110)
      (push 'L130)
      (push 'L150)
      (do_Z 2 1 0)
      (do_top)
      L90 (do_I)
      (do_top)
      L110 (push 4)
      (pexit)
      L130 (push 3)
      (pexit)
      L150 (do_I)
      (do_top))))
```

Evaluation Steps:

```
main_program Example_6
processors=128
(stack= (L10))
(ret_stack= (done))
(do_top)
(stack= nil)
```
(ret_stack= (done))
(push L30)
  (stack= (L30))
  (ret_stack= (done))
(push L50)
  (stack= (L50 L30))
  (ret_stack= (done))
(push L70)
  (stack= (L70 L50 L30))
  (ret_stack= (done))
(push L90)
  (stack= (L90 L70 L50 L30))
  (ret_stack= (done))
(do_ Y 2 1 2 (Proc_2) (Proc_3))
  (stack= ((L90 L30) Proc_2 (timeflag (4 2) 1)))
  (ret_stack= (done))
(do_top)
  (stack= (L30 Proc_2 (timeflag (4 2) 1)))
  (ret_stack= (done))
(do_ 1)
  (stack= (L30 Proc_2 (timeflag (4 2) 1)))
  (ret_stack= (done))
(do_top)
  (stack= (L31 Proc_2 (timeflag (4 2) 1)))
  (ret_stack= (done))
(call1 L31)
  (stack= (L31 Proc_2 (timeflag (4 2) 1)))
  (ret_stack= (done))
(do_top)
  (stack= (L32 Proc_2 (timeflag (4 2) 1)))
  (ret_stack= (done))
(call2 L32)
  (stack= (L32 Proc_2 (timeflag (4 2) 1)))
  (ret_stack= (done))
(do_top)
  (stack= ((timeflag (12 3) 7) 1))
  (ret_stack= (done))
(wait 1)
  (stack= (L32 (timeflag (12 3) 7) 1))
  (ret_stack= (done))
(do_top)
  (stack= (7 1))
  (ret_stack= (done))
(wait 12)
  (stack= (7 1))
  (ret_stack= (done))
(do_plus)
  (stack= (8))
  (ret_stack= (done))
(pexit)
Proc_2
  (L70 L30)
  (stack= (L30))
(ret_stack= (nil))
(push L110)
(stack= (L110 L30))
(ret_stack= (nil))
(push L130)
(stack= (L130 L110 L30))
(ret_stack= (nil))
(push L150)
(stack= (L150 L130 L110 L30))
(ret_stack= (nil))
(do Z 2 1 0 (Proc_4) (Proc_5))
(stack= ((L150 L30) (timeflag (4 2) 3) (timeflag (4 2) 4)))
(ret_stack= (nil))
(do_top)
(stack= (L30 (timeflag (4 2) 3) (timeflag (4 2) 4)))
(ret_stack= (nil))
(do_J)
(stack= (L30 (timeflag (4 2) 3) (timeflag (4 2) 4)))
(ret_stack= (nil))
(do_top)
(stack= (L31 (timeflag (4 2) 3) (timeflag (4 2) 4)))
(ret_stack= (nil))
(call L31)
(stack= (L31 (timeflag (4 2) 3) (timeflag (4 2) 4)))
(ret_stack= (nil))
(do_top)
(stack= (L32 (timeflag (4 2) 3) (timeflag (4 2) 4)))
(ret_stack= (nil))
(call L32)
(stack= (L32 (timeflag (4 2) 3) (timeflag (4 2) 4)))
(ret_stack= (nil))
(do_top)
(stack= (3 4))
(ret_stack= (nil))
(wait 1)
(stack= (3 4))
(ret_stack= (nil))
(do_plus)
(stack= (7))
(ret_stack= (nil))
(pexit)
(7)

Proc_5  (L110)
(stack= nil)
(ret_stack= (nil))
(push 4)
(stack= (4))
(ret_stack= (nil))
(pexit)
(4)

Proc_4  (L130)
(stack= nil)
(ret_stack= (nil))
(push 3)
(stack= (3))
(ret_stack= (nil))
pexit
(3)

Proc_3 (L50)
(stack= nil)
(ret_stack= (nil))
(push 1)
(stack= (1))
(ret_stack= (nil))
pexit
(1)
5. PERFORMANCE MEASUREMENTS

As with any new system, we are interested in how well ours performs. Our implementation allows us to study the performance of not only the single processor setup, but multiprocessor cases as well. We want to be able to answer questions such as:

Will adding additional processors be beneficial?
How many processors should be used?
How much faster will my program run?
How much does it cost to add more processors?

Before presenting some performance results of the examples presented earlier, we will first discuss some traditional measures and then talk about the additions to our system that allow us to collect the statistics necessary to calculate some measures.

5.1 Traditional Measures

There are various sources on measuring performance, but one treatment that we have decided to include in our study is presented by Hwang and Briggs [8b]. They suggest calculating several performance measures to compare a parallel computation with a serial computation. Let us denote

\[ n = \text{number of processors} \]
\[ P_a = \text{total number of operations to be performed} \]
\[ T_a = \text{total execution time in steps by the system}. \]

We will assume that each operation requires one step of execution time, so that \( T_1 = P_1 \) for a sequential evaluation. When a program is executed in parallel, we can observe that the total execution time is less than the total number of steps needed to evaluate the program \( (T_n < P_n) \). This occurs because more than one operation can be executed during
the same time interval. The following performance indices are suggested:

1. **Speedup**

   \[ S_n = \frac{T_1}{T_n} \]

   The speedup of a system indicates how much faster a parallel computation is compared to a sequential computation of the same program.

2. **Efficiency**

   \[ E_n = \frac{T_1}{n \times T_n} \]

   When we speak of the efficiency of a system, we are interested in how productive the system is as a unit. We want little waste of processor capability during evaluation of a program.

3. **Redundancy**

   \[ R_n = \frac{P_n}{P_1} \]

   The redundancy index indicates how much extra work may be required due to the multiprocessor system as compared to the work required by a single processor solution. This extra work may arise from more than one processor evaluating the same operations. Since \( 1 \leq R_n \), the redundancy measure can be a guide to some of the overhead incurred by using a multiprocessor solution.

4. **Utilization**

   \[ U_n = \frac{P_n}{n \times T_n} \]

   The utilization factor can be used to denote how effectively the system is using its processors during evaluation of a program. The utilization index is a factor of the efficiency and the redundancy of the system since

   \[ U_n = R_n \times E_n. \]
5. Quality

\[ Q_n = \frac{T_1^3}{n \times T_n^4 \times P_n} \]

The quality index, as defined above, is rather elusive. However, an equivalent definition in terms of the speedup, efficiency, and redundancy of the system is given below.

\[ Q_n = \frac{S_n \times E_n}{R_n} \]

The measure indicates how well the system performs, given the fact that the multiprocessor setup may have to execute more operations than a uniprocessor serial solution.

These measures may or may not be meaningful within our framework. They are suggestive of what can be defined and, of course, are easily obtainable from the statistics we are gathering.

5.2 Measures of Our System

Once we verified that our implementation was working and that we could specify the number of processors available to evaluate a particular program, we wanted to answer some of the questions we listed at the beginning of this chapter. Was our multiprocessor implementation any better than the uniprocessor serial evaluation? To answer such questions, we added the capability to collect statistics during evaluation.

1. Various counters were introduced to keep running totals of the operations performed. Every machine instruction has a corresponding counter. For example, do_newS is tallied in the counter newSctr, and topctr keeps track of the number of times do_top is executed. One surprise these counters gave us was to inform us that in some examples fewer combinators are evaluated in a multiprocessor setup than in the
uniprocessor case. We will present such an example later in this chapter.

2. The next step we took in measuring the performance of our system was to estimate processor time. Since we are simulating a multiprocessor system, we couldn't actually measure the CPU time for each processor. We decided that each machine instruction would take one processor "tick" of time. Thus, whenever an instruction is executed, the current processor's "clock" is updated. We realize that some instructions are more complicated than others, but we feel this method provides us with a starting place in evaluating processor time.

3. We introduced a measure of the idle time of each processor. Our algorithm in the multiprocessor implementation forces a processor to wait for results from a child processor rather than continue to process with the child's id. If a processor enters a busy wait situation, we wanted to know how much processor time was being used waiting for results from its child processors. Each time a processor finds holes in its stack and is prevented from continuing with evaluation, the idle time is incremented.

4. Each time a child processor is allocated, some amount of overhead is incurred. This extra time results from the messages that must be passed between parent and child. Our system keeps these messages to a minimum. All that is passed is an initial stack to a child and the results returned to the parent. We wanted to provide the option of charging for this overhead. Additionally, we have added the capability to vary the amount charged for the overhead resulting from the message passing. The overhead_constant indicates the amount charged for each message. This amount remains constant regardless of the length of the message. If we choose to ignore the overhead, we merely initialize the overhead_constant to zero. Since our performance studies are in their initial stages, we are satisfied with this arrangement for now. This is an area for future development.
5. Some miscellaneous measures have also been included out of our interest in observing them. Included in this category is the maximum number of processors executing in parallel during the course of program evaluation. This number provides us with information in determining an "optimal" multiprocessor setup when an infinite (relatively speaking) number of processors are not available. Another statistic we collect is the total number of forks and joins executed. These numbers should be equal, since every child processor allocated should also be deallocated after it finishes its work. The number of forks tells us how many child processors are used in the course of evaluating a program. It is interesting to compare this number with the maximum number of processors executing in parallel, and the total number of processors in the system. When the number of forks is greater than the total number of processors in the system, we can tell that processors are being used more than once. We will explore some of these relationships later in some examples.

5.3 Interpretation of Our Measures

Each processor in our system has associated with it a time value representing the number of "ticks" required for evaluation of its assigned computation. In the simplest framework involving no overhead or idle time, the number of ticks equals the number of operations the processor must execute. In general, when idle time and/or overhead are present, adjustments to this time are made depending on the values of the idle time of the processor and the overhead incurred.

The following examples will clarify how time is assessed to a processor. We will assume an operation requires one tick and that the overhead constant has been determined to be equal to two (one tick for the initial stack sent to the child plus one tick for the child to return its results back to the parent).
1. This first example demonstrates the situation where the parent processor allocates two child processors. The parent continues to evaluate code and is not forced to become idle while waiting for results from its children. The results are available for the parent processor to use before the parent needs them. In this example, there are two generations of processors. Neither of the child processors allocates new child processors. Note: \* = 1 tick.

   processor 1 |**********|
   processor 2 |* * * * |
   processor 3 |* * |

Processor 2 required four ticks of processing time plus its overhead charge of two. Processor 3 required two ticks of processing time and a charge of two ticks of overhead. The parent processor (l), took ten ticks to complete its evaluation. To calculate its overhead charge, the formula is as follows:

   \[ \text{overhead} = \min((\max(\text{overhead}_{\text{proc2}}, \text{overhead}_{\text{proc3}}) - \text{idletime}), 0) \]

   \[ \text{overhead} = \min((\max(2.2) - 0), 0) \]

2. The next example demonstrates a processor forking off a child and experiencing idle time. The child also allocates another child processor.

   processor 1 |* * * * * w w * * |
   processor 2 |* w w w * |
   processor 3 |* * * |

Processor 3 requires three units of processing time plus has an overhead of two associated with it. The total processing time for processor 2 is five, with three of those ticks assessed to idletime. The overhead of this processor includes the overhead associated with its messages from and to the parent, as well as the overhead assessed to its child. Therefore, the total overhead is four. Processor 1 uses ten units of time, including idle time, and has an associated overhead of four.
5.4 An In-Depth Interpretation of Our Measures

While the samples given above are somewhat useful, it is more meaningful to study an actual example program. Example 7 was chosen because it provides us with a medium number of processor allocations and several different scenarios to illustrate how processor time, idle time, and overhead are calculated.

If we examine the trace of Example 7, we can tell how many operations are executed, when child processors are allocated, and how long each processor must wait. We can use the trace information we looked at in Chapter 3, but remove all the extra information about the stack values, the busy wait do_tops, etc. What remains is a list of the machine instructions that get executed. The trace information for a case charging overhead will provide us with the same operation list as the case that charges no overhead. Therefore, the following list indicates the differences in the wait times (idle time) by giving the wait time for the charged overhead case first, and denoting the no overhead case by a "*".

The following list is obtained in this manner for the main processor:

```
main_program Example_7
  (do_top)
  (push L30)
  (push L50)
  (push L70)
  (do_X 1 1 1 (Proc_2))
  (do_top)
  (push L110)
  (push L150)
  (wait 10) (* wait 6)
  (do_newS 1 1 (Proc_4))
  (do_top)
  (do_1)
  (do_top)
  (wait 13) (* wait 8)
  (do_Z 2 1 1 (Proc_7))
  (do_top)
  (call1 L191)
  (do_top)
  (call2 L192)
  (do_top)
  (wait 3) (* wait 1)
  (do_plus)
```
An explanation is needed about the call1, do_top, call2, do_top, do_plus sequence that we find in most of our examples. The call1 and call2 instructions are included only for our handling of binary operators. Thus, we do not think that a binary operation should be charged for the four ticks accumulated for executing the call1, do_top, call2, do_top sequence. All we really want to do is charge for the do_plus. So in counting the operations performed, we will only count the do_plus instruction within the above-mentioned sequence. Also, when do_top appears in the trace for the busy wait situation, none of these count toward the time tally. The idle time takes care of this extra time. We have removed the unnecessary do_top, but have left in the call sequence for clarity purposes.

A timing summary for processor 1 (the main processor) is useful in interpreting the sequence of events during execution. All of the processors' execution time summaries will be given in a table form later in this section. Processor 1 starts execution and proceeds to execute five instructions. At this time, it allocates processor 2. Processor 1 continues to work for three more ticks, and then enters a wait state for 10(*6) counts. Upon receiving necessary information from processor 2, it executes one more instruction and then starts up processor 4. Continuing for three more units of time, it then finds it needs more information from processor 4. Processor 1 waits for 13(*8) ticks, receives the information, executes one more instruction, and starts up processor 7. The main processor executes another instruction (recall we ignore the call1, etc.), waits for 3(*1) more units, and finally executes its final two instructions. The total time spent in executing was 16 ticks and an additional 26(*15) time units was spent waiting. Thus, the total execution time was 42(*31) ticks. This summary ignored the overhead associated with the various child processors, but we will discuss that in the next section.

The following are summaries for each of the other processors:
Proc_2
(push L90)
(push L130)
(push L190)
(push L230)
(do_X 3 1 3 (Proc_3))
(do_top)
(wait 5) (* wait 3)
(spflag (do_Z 2 1 1) L190 L130 1)

Proc_3
(do_J)
(do_top)
(push 1)
(pexit)
(1)

Proc_4
(push L170)
(push L210)
(do_Y 1 1 1 (Proc_5))
(do_top)
(do_J)
(do_top)
(wait 1) (* wait 0)
(do_Z 2 1 1 (Proc_6))
(do_top)
(call1 L191)
(do_top)
(call2 L192)
(do_top)
(wait 3) (* wait 1)
(pexit)
(3)

Proc_5
(push 2)
(pexit)
(2)

Proc_6
(do_J)
(do_top)
(2)

Proc_7
(do_J)
(do_top)
(3)
Example 7 Summaries, No Overhead Charged

<table>
<thead>
<tr>
<th>Processor</th>
<th>Parent</th>
<th>Times</th>
<th>Idle Time</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>31</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>9</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>11</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Example 7 Summaries, Overhead Charged

<table>
<thead>
<tr>
<th>Processor</th>
<th>Parent</th>
<th>Times</th>
<th>Idle Time</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>42</td>
<td>26</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>13</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>16</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

In comparing the above two tables of summaries, we can see that the following relationships hold for each of the child processors:

1. number of operations = time_{no_{ovhd}} - idle time_{no_{ovhd}}
2. time_{ovhd} = number of operations + idle time_{ovhd} + overhead constant

At first glance, it may seem odd that the idle time value is greater in the case where overhead is charged than the value when no overhead is charged. This results because of the overhead associated with starting up the child processors. When overhead is being charged, it takes longer for the parent to get its results back, therefore its idle time is longer. If no overhead is charged, the results arrive instantaneously from the child.

The following is a timing trace of Example 7. This particular trace is for the situation where overhead is charged. The meaning of the "t" is one tick of execution time, "w" indicates one tick of wait time, and a number (2 through 7) indicates the allocation of that particular processor.
We think this time diagram needs some explanation, since it is the first one presented.

Each processor's activity and timings will be explained to help clarify the interactions between processors.
Processor 1: Executes for 5 ticks, starts processor 2, executes for 3 ticks, and waits for results back from processor 2.

Processor 2: Executes for 5 ticks, starts processor 3, executes for 1 tick, doesn't have enough parameters to continue, so waits for results from processor 3 before it can return its results (partially evaluated code) back to processor 1.

Processor 3: Executes for 4 ticks and returns its results to processor 2.

The execution time for processor 3 is 4, but there is overhead involved, so we must add an additional 2, bringing the total time to 6. Thus, processor 2, after starting processor 3 and executing for 1 tick, must wait the difference between the time its child is taking and the time it is able to execute in parallel (6—1) = 5.

Processor 2: After receiving results from processor 3, this processor is ready to return its results to processor 1. The total number of operations executed is 6, its idle time is 5, and its overhead charges are 4 (2 for its child and 2 for itself). Therefore, the total time charge is (6+5+2) = 13. (Recall the formula calls for adding the number of operations plus the idle time plus the overhead constant charge for this processor's messages from and to its parent.)

Processor 1: The idle time for this waiting period is 10 since the processor was able to execute for 3 ticks in parallel with its child, processor 2, and the time charged to processor 2 was 13. Now, processor 1 uses 1 tick, starts up processor 4, executes for 3 ticks, and again must wait for results from processor 4.

Processor 4: Executes for 3 ticks, starts up processor 5, executes for 3 ticks, and waits for results from processor 5.
Processor 5: Executes for 2 ticks, and returns its results to processor 4. Total time is 4 (2 ticks execution plus 2 ticks overhead).

Processor 4: The total time processor 4 spent waiting for results from processor 5 was 4—3 = 1 tick. Processor 4 continues to execute for 1 tick, starts up processor 6, executes for 1 tick, and waits for results from processor 6.

Processor 6: Executes for 2 ticks and returns. Total time is 4 (including overhead).

Processor 4: Waiting time calculates to be 3, executes for 2 ticks, and returns its results to processor 1. This processor executed 10 operations, had 4 units of wait time, and has an associated overhead of 6 (its own 2, plus 2 for each of its child processors). Total time is 16.

Processor 1: The total time it spent waiting for the results from processor 4 was 16—3 = 13. Processor 1 executes 1 more tick, starts up processor 7, executes 1 more tick, and finds it must wait for results from processor 7.

Processor 7: Executes for 2 ticks and returns. Total time is 4.

Processor 1: Total wait time was 4—1 = 3. Completes its evaluation after 2 more ticks.

The total number of operations performed by processor 1 was 16, it was idle for a total of 26 ticks. Therefore, its total processor time was 42. The total overhead charge for the parent processor is calculated as follows: When a child processor reports back its results, it also reports back its total overhead charges. The parent processor keeps a running total of overhead incurred for all of its children. However the amount added to the running total is the difference between the amount of overhead reported by the child and the
amount of parallel execution time of the parent.

We included the performance analysis phase in our work because we wanted to answer several questions about our multiprocessor implementation. Is the multiprocessor evaluation any better than a uniprocessor evaluation? We think so, based on the statistics we have collected. The total execution time for the multiprocessor setup is shorter than the time a single processor system requires. Even with our overhead charges, it appears the multiprocessor system requires a shorter amount of time.

Our investigation into the performance of our multiprocessing methodology has just begun, but is an area that we would like to continue to pursue. In the very near future, we will be collecting the statistics to help us answer those questions listed in the beginning of this chapter:

Will adding additional processors be beneficial?

How many processors should be used?

How much faster will my program run?

How much does it cost to add more processors?
6. SUMMARY

6.1 Thesis Summary

Our belief is that computer architectures should be derived from the programming languages they are intended to support. We have demonstrated one approach to analyzing a functional language that will indicate the underlying architecture required for support of that language. Furthermore, we contend that the inherent parallelism of functional programming languages should automatically be supported by such a derived architecture. Our methodology focuses on the semantics of the language. The parallelism is inherent to the precise definition of the language, therefore our translation into an equivalent combinator representation preserves this parallelism. The combinators have been shown to dictate the supporting architecture, and also to provide a built-in mechanism for decomposition of programs, thereby directing the allocation of parallel evaluation.

Influenced by the work of Turner and Abdali, we developed a new set of combinators. The improvements in our combinators combine the desirable features of both methods. Our combinators support simultaneous definitions in the lambda calculus and the use of multiple parameters in functions. Turner's method provided us with the insight necessary to introduce optimization rules for our combinators. These optimization rules allowed us to develop code that was very competitive, and in some sense, better than the code produced by either Turner's or Abdali's methods.

Our combinators prove to be dual-purposed. Not only do they serve as machine instructions for our underlying evaluator, but they can be used as control mechanisms for parallel evaluation. The combinators manage the decomposition of our program and allocate the pieces to processors for parallel evaluation.
More important than the combiners we developed, we have established a methodology for generating new combiners. This will allow us, at a later date, to tackle the problem of finding an "optimal" set of combiners. We have thus provided a method to tailor, or adjust, a multiprocessor system to utilize the inherent parallelism in programs during evaluation.

Once we established that our methodology did indeed provide us with the mechanism to specify the supporting architecture and also provide the control mechanism for parallel evaluation, we wanted to start the implementation phase of our project. This is a continuing effort, but we are quite happy with results of our fixed-program machine. We have been able to use the combiners as machine instructions, and also have been able to automatically decompose our programs for evaluation on multiple processors.

We are now in the beginning stages of evaluating the benefits of our multiprocessing scheme. We need to assess the grain of parallelism that is being supported by our current set of combiners. Performance statistics are encouraging so far. As mentioned above, we should be able to tune our system via the introduction of different combiners. We believe our method will lead to an implementation that will be highly competitive with other parallel machines.

6.2 Future Research

While this thesis is the culmination of many efforts, we feel that in many ways, we have just opened the door to many exciting areas of future research. Some of the projects we would like to work on are merely adjustments to our current methodology. Other areas are enhancements and extensions to our project. Of course, the most difficult direction we will need to explore is how our efforts fit into the overall picture of multiprocessing.
We now present a few of the shorter term goals we will be exploring in the near future:

1. We anticipate the need to study a great number of programs in order to collect the performance measurement statistics needed to make comparisons between our methods and other known sequential and parallel implementations. Currently, we have been doing all the abstractions on the lambda calculus expressions by hand in order to generate our combinator representations. We plan to develop a compiler to automatically perform the translation from lambda calculus to combinator representations. This will be a relatively simple job. We have previously implemented such a compiler to translate lambda calculus expressions into Turner's combinator representations. What we need to do is modify this compiler to handle our new set of combinators and abstraction rules.

2. In this same line of thought, we will want to automatically generate the code for our evaluator from the combinator sequences. Again, this will merely be an extension to some previous work. We currently have a code generator that handles Turner's combinators and produces code suitable for Jones' machine. The modifications will be fairly simple to allow the generator to recognize our new set of combinators.

3. We will continue in our performance measurement efforts. In particular, we desire to refine our measurement statistics. We have made a good start on what we want to measure and what we think represents valuable performance indicators. We need to work through the problem very carefully to see if we can come up with a really good set of performance indicators for multiprocessing systems.

We have a very powerful implementation that will allow us to study the effects of adding and removing processors. Most of our work to date has been with an infinite number of processors. We will want to take advantage of our capability to specify finite numbers of processors and carry out our performance measurements in these situations also.
Once we have the performance measurement strategies in place, we will want to explore what impact some of our policy decisions have on the performance of the system.

4. We will want to investigate new methods of processor management. In particular, we have implemented various decisions as to how the parallelism is allocated. Currently, a processor is responsible to perform its own work if there are not enough processors. We would like to modify the algorithm to retry the allocation procedure. Our current method allocates pieces of code from the top of the stack. We would like to investigate the implications of this, and also modify the allocation scheme to start with the tail end of the stack in order to optimize the amount of work the parent can perform before having to wait for results from its child processors.

As we indicated, we need to discover how our methods compare to other implementation strategies. This is a longer term goal.

5. We really need to develop a set of benchmark programs. A few exist for Lisp systems and these will be a good start. However, few such benchmarks efforts exist for multiprocessor systems. We need to develop a good set, and then proceed to use it in a thorough evaluation of our approach in comparison with purely sequential approaches (like Turner's).

6. One obvious area we will want to explore is the tuning of the architecture via the introduction of alternative combinators. This project started as an investigation of alternative architectures for the support of functional languages. As we mentioned in our conclusions, we can work towards finding an optimal set of combinators that will dictate an architecture best suited for the support of a particular functional language.

7. Finally, we would like to build our machine. We feel it will be feasible to do with off-the-shelf microprocessors and we believe such a prototyping effort will be worthwhile.
7. REFERENCES


I have come to realize that the completion of this dissertation does not represent the final product, but is the result of a resting point in my journey. I have lost sight of what I once considered to be a final hurdle as I now am focused on what lies ahead. Thanks Charlie, for consistently reminding me of the larger picture.

I would like to express appreciation for the guidance and support I received during my experiences at Iowa State. In particular, I thank my committee members for the time and effort they spent reviewing my work: Charles T. Wright, Jr., who directed my research; Arthur V. Pohm, who also served as my co-major professor; Dale Grosvenor; M. Prabhu; Terry Smay; and Robert M. Stewart.

I am fortunate to have received technical advice and, more important, moral support from Jim Davis and Diane Rover. Each in his/her own way helped me endure the storms. Thank you.

My warmest thanks are to my family, especially to my parents. Their words of encouragement lifted my spirits, especially during the final stages when I felt so isolated from life. Finally, I would like to express my heartfelt appreciation of my husband, Dan. His acceptance of who I am and what I strive to do, and his caring support have carried me through many years.
This appendix contains the compilation rules used by Jones and Muchnick to translate combinator expressions (using Turner's combinators) into machine code for a simple single processor computer. Code, written in Franz Lisp, defining the actions of the computer is also given.

**Compilation Rules**

A label \( L_p \) is assigned to each node \( p \) corresponding to \( expr_1, expr_2, ..., expr_n \) that define the functions \( f_1, f_2, ..., f_n \). Given that \( p_1, p_2, ..., p_n \) represent the root nodes of the corresponding expressions trees for these expressions, the following code gets generated for a program:

\[
\begin{align*}
\text{(bind 'f1 'Lp1)} \\
\vdots \\
\text{(bind 'fn 'Lpn)} \\
\text{(first_call 'done)} \\
\text{done (finish)} \\
\text{Lp1 (Code(p_1))} \\
\vdots \\
\text{Lpn (Code(p_n))}
\end{align*}
\]

where Code is defined below (with \( p, q, \) and \( r \) representing nodes):

\[
\begin{align*}
\text{Code}(p \ q \ r) &= \text{Lp (push 'Lr0)} \\
&\quad \text{Code}(q) \\
\text{Code}(p \ X) &= \text{Lp (do_X)} \\
&\quad \text{(do_top) where X is any of the combinators I, K, S, B, or C} \\
\text{Code}(p \ c) &= \text{Lp (push c)} \\
&\quad \text{(pexit) where c is any constant} \\
\text{Code}(p \ op) &= \text{Lp (call1 'Lp1)} \\
&\quad \text{Lp1 (call2 'Lp2)} \\
&\quad \text{Lp2 (do_op)} \\
&\quad \text{(pexit) for any binary operator op} \\
\text{Code}(p \ cond) &= \text{Lp (call_cond 'Lp1)} \\
&\quad \text{Lp1 (do_cond)}
\end{align*}
\]
Code($p f_i$) = $L_p$ (lookup $'f_i'$)

(\textbf{Defining Machine Code})

\begin{verbatim}
(def bind
  (lambda (fname action)
    (setq env (cons (cons fname action) env))))
(def push
  (lambda (action)
    (setq stack (cons action stack))))
(def look_up
  (lambda (name)
    (cond ((null (assoc name env))
      (progn (print (list "undefined 'function: name))
        "badfunction))
    (t (setq stack
      (cons (cdr (assoc name env)) stack))))))
(def first_call
  (lambda (pexit)
    (setq ret_stack (list pexit))
    (do_top)))))
(def do_top
  (lambda ()
    (check_args 0 'topctr)
    (cond ((numberp (slkl)) (go (no_quote (retslkl))))
      ((atom (slkl))
        (progn (setq tem (slkl))
          (setq stack (stklr))
          (go (no_quote tem))))
      (t
        (progn (setq oldpair (stkl))
          (setq stack (cons (caar slack)
               (cons (cdar slack) (stklr))))
          (do_top))))))
(def stk1 (lambda () (car stack)))
(setq stack '(2))
(setq ret_stack '())
(def stk2 (lambda () (cadr stack)))
(def stk3 (lambda () (caddr stack)))
(def stk1r (lambda () (cdr stack)))
(def stk2r (lambda () (cddr stack)))
(def stk3r (lambda () (cdddr stack)))
(def retstk1 (lambda () (car ret_stack)))
(def retstk2 (lambda () (cadr ret_stack)))
(def retstk3 (lambda () (caddr ret_stack)))
(def retstk1r (lambda () (cdr ret_stack)))
(def retstk2r (lambda () (cddr ret_stack)))
(def retstk3r (lambda () (cdddr ret_stack)))
(def check_args
  (lambda (n counter)

\end{verbatim}
(cond ((< (length stack) n)
  (progn (setq stack (cons tern stack))
    (go (no_quote (retstk1))))
  (t (set counter (1+ (eval counter)))))))
(def result
  (lambda (n counter)
    (set counter (1+ (eval counter)))
    (setq stack (cons n (retstk3)))
    (setq ret_stack (retstk3r))))
(def pexit
  (lambda ()
    (go (no_quote (retstk1))))))
(def no_quote
  (lambda (lab)
    (cond ((atom lab) lab)
      (t (cond ((eq (car lab) quote) (no_quote (cdr lab)))
        (t (error (list 'badgoto: lab))))))))
(def start
  (lambda (n)
    (setq topctr 0 Sctr 0 Kctr 0 lctr 0 Bctr 0
      Cctr 0 plusctr 0 minusctr 0 timesctr 0
eqctr 0 condctr 0 binopctr 0)
    (setq stack (list n))
    (setq env nil)))
(def finish
  (lambda ()
    (print (cons (stkl)
      (list
        'lop= topctr
        'S= Sctr
        'K= Kctr
        'l= lctr
        'B= Bctr
        'C= Cctr
        'plus= plusctr
        'minus= minusctr
        'times= timesctr
        'eq= eqctr
        'cond= condctr))))
(reset)))
(def call1
  (lambda (pexit)
    (check_args 2 binopctr)
    (setq ret_stack (cons pexit (cons (stklr) ret_stack)))
    (setq stack (list (stkl)))
    (do_top)))
(def call2
  (lambda (pexit)
    (setq tem ret_stack)
    (setq ret_stack
      (cons pexit (cons (stkl1)
(cons (cdadr ret_stack) (retstk2r)))))
(setq stack (list (caadr tem)))
(do_top))

(defun do_cond
  (lambda ()
    (setq stack
      (cons (cond ((stkl) (caadr rel_slack))
        (t (cadadr ret_stack)))
        (cddadr ret_stack)))
    (setq ret_stack (retstk2r))
    (do_top))

(defun call_cond
  (lambda (pexit)
    (check_args 3 condctr)
    (setq ret_stack (cons pexit (cons (stklr) ret_stack)))
    (setq stack (list (stkl)))
    (do_top))

(defun do_I
  (lambda ()
    (check_args 1 'lctr))

(defun do_K
  (lambda ()
    (check_args 2 'Kctr)
    (setq stack (cons (stkl) (stk2r)))))

(defun do_S
  (lambda ()
    (check_args 3 'Sctr)
    (setq stack
      (cons (stkl) (cons (stk3) (cons (cons (stk2) (stk3r)) (stk3r)))))))

(defun do_B
  (lambda ()
    (check_args 3 'Bctr)
    (setq stack (cons (stkl)
      (cons (cons (stk2) (stk3r)) (stk3r))))))

(defun do_C
  (lambda ()
    (check_args 3 'Cctr)
    (setq stack (cons (stkl)
      (cons (stk3) (cons (cons (stk2) (stk3r)) (stk3r))))))

(defun do_times
  (lambda ()
    (setq tem (retslk2))
    (result (* tem (stkl)) 'timesctr))

(defun do_plus
  (lambda ()
    (setq tem (retslk2))
    (result (+ tem (stkl)) 'plusctr))

This appendix contains code, written in Franz Lisp, for a multiprocessor computer that will evaluate combinator expressions containing extended combinators.

(def bind)
  (lambda (fname action)
    (setq env (cons (cons fname action) env)))

(def push)
  (lambda (action)
    (check-args 0 pushcptr)
    (my-trace (list 'push action))
    (setq stack (cons action stack)))

(def my_push)
  (lambda (action)
    (setq stack (cons action stack)))

(def look_up)
  (lambda (name)
    (cond ((null (assoc name env))
      (progn (print (list 'undefined 'function: name))
       'badfunction))
      (t (setq stack
        (cons (cdr (assoc name env)) stack)))))

(def first_call)
  (lambda (cont)
    (setq ret_stack (list cont))
    (do_top)))

(def stk1 (lambda () (car stack)))
(def stk2 (lambda () (cadr stack)))
(def stk3 (lambda () (caddr stack)))
(def stk1r (lambda () (cdr stack)))
(def stk2r (lambda () (cddr stack)))
(def stk3r (lambda () (cdddr stack)))
(def retstk1 (lambda () (car ret_stack)))
(def retstk2 (lambda () (cadr ret_stack)))
(def retstk3 (lambda () (caddr ret_stack)))
(def retstk1r (lambda () (cdr ret_stack)))
(def retstk2r (lambda () (cddr ret_stack)))
(def retstk3r (lambda () (cdddr ret_stack)))
(def check_args)
  (lambda (n counter)
    (cond ((< (length stack) n)
      (progn (setq stack (cons tem stack))
       (go (no-quote (retstk1))))
      (t (inc_clock)
       (set counter (1+ (eval counter)))))
    (def check_only)
      (lambda (n)
(cond ((< (length stack) n)
    (progn (setq stack (cons n (retstk3)))
    (go (no_quote (retstk1)))))

(def result
    (lambda (n counter)
        (setq counter (1+ (eval counter)))
        (inc_clock)
        (setq stack (cons n (retstk3)))
        (setq ret_stack (retstk3))))

(def pexit
    (lambda ()
        (check_args 0 'pexit)
        (my_trace (list 'pexit))
        (go (no_quote (retstk1)))))

(def no_quote
    (lambda (lab)
        (setq top_temp lab)
        (cond (null lab) (join)))
        (cond (atom lab) lab)
        (t (cond ((eq (car lab) 'quote)
                    (no_quote (cdr lab)))
                (t (error (list "badgoto: lab")))))

(def start
    (lambda ()
        (setq pushctr 0 topctr 0 nev 0 opctr 0 Stctr 0 Kctr 0 Ictr 0 Bctr 0
            Xctr 0 Yctr 0 Zctr 0 newSctr 0 newKctr 0
            newlctr 0 newKctr 0
            newSctr 0 newKctr 0
            newlctr 0 newKctr 0
            newSctr 0 newKctr 0
            newlctr 0 newKctr 0)
        (setq masler_clock 0)
        (setq current_trace nil)
        (setq clock_ticks 0)
        (setq grand_master 0)
        (setq stack nil)
        (setq env nil)))

(def numprocs
    (lambda (x)
        (setq avail_procs x)
        (setq avail_procs x)
        (setq avail_procs x)))

(def finish
    (lambda ()
        (update_clocks)
        (terpri print_file)
        (terpri print_file))

(print prog_name print_file)
(terpri print_file)
(print
  (list
    'stktop= stk1
    'master_clock= master_clock
    'max_procs= max_in_use
    'idle_time= idle_time
    'overhead_constant= overhead_constant
    'overhead= overhead) print_file)
(terpri print_file)
(tab 10 print_file)
(print
  (list
    'pexit= pexitctr
    'push= pushctr
    'top= toptcr
    'newtop= newtopctr
    'S= Sctr
    'K= Kctr
    'I= lctr
    'B= Bctr
    'C= Cctr
  ) print_file)
(terpri print_file)
(tab 10 print_file)
(print
  (list
    'X= Xctr
    'Y= Yctr
    'Z= Zctr
    'newS= newScr
    'newI= newlctr
    'newK= newKctr
  ) print_file)
(terpri print_file)
(tab 10 print_file)
(print
  (list
    'plus= plusctr
    'minus= minusctr
    'times= timesctr
    'eq= eqctr
    'fork= forkctr
    'join= joinctr
    'binop= binopctr
    'cond= condctr
  ) print_file)
(terpri print_file)
(print times= print_file) (terpri print_file)
(print time_counts print_file) (terpri print_file)
(print idles= print_file) (terpri print_file)
(print idle_summaries print_file) (terpri print_file)
(print 'overhead= print_file)
(def do_cond
  (lambda ()
    (setq stack
          (cons (cond ((stkl) (caadr ret_stack))
                 (t (cadadr ret_stack)))
               (cddadr ret_stack)))
    (setq ret_stack (retstk2r))
    (do_top)))

(def call_cond
  (lambda (cont)
    (check_args 3 'condcont)
    (my_ltrace (list cond cont))
    (setq ret_stack (cons cont (cons (stklr) ret_stack)))
    (setq stack (list (stkl)))
    (do_top)))

(def do_I
  (lambda ()
    (setq idle_temp (max clock_ticks (untag_stack 1)))
    (cond ((neq idle_temp clock_ticks) (setq overhead (+ overhead
                       (min local_overhead (- idle_temp clock_ticks)))))
           (setq idle_time (+ idle_time (- idle_temp clock_ticks))))
    (cond ((neq idle_temp clock_ticks) (my_ltrace (list
                   'wait (- idle_temp clock_ticks))))
           (setq clock_ticks idle_temp))
    (cond ((> (hole_count 1 1) 0))
          (my_push (list 'do_I))
          (my_push resumeflag)
          (my_top))
    (t
     (cond ((< (length stack) 1)
              (my_push (list 'do_I))
              (my_push resumeflag)
              (join)))
     (check_args 1 'lctr)
     (my_trace (list 'do_I))
     (cond (resumed (setq resumed nil) (do_top))))))

(def do_K
  (lambda ()
    (check_args 2 'Kctr)
    (my_trace (list 'do_K))
    (do_top)))
(setq stack (cons (stk1) (stk2r))))

(defun do_S
  (lambda ()
    (check_args 3 'Sctr)
    (my_trace (list 'do_S))
    (setq stack
      (cons (stk1) (cons (stk3) (cons (stk2) (stk3r) (stk3r))))))

(defun do_B
  (lambda ()
    (check_args 3 'Bctr)
    (my_trace (list 'do_B))
    (setq stack (cons (stk1) (cons (stk2) (stk3r) (stk3r))))))

(defun do_C
  (lambda ()
    (check_args 3 'Cctr)
    (my_trace (list 'do_C))
    (setq stack (cons (stk1) (cons (stk3r) (stk3r) (stk3r))))))

(defun toss
  (lambda (n 1)
    (cond
      ((eq n 0) 1)
      (t (toss (- n 1) (cdr 1))))))

(defun find
  (lambda (n 1)
    (cond
      ((eq n 0) nil)
      ((eq n 1) (car 1))
      (t (find (- n 1) (cdr 1))))))

(defun extract
  (lambda (s n 1)
    (firstn n (toss (- s 1) 1))))

(defun firstn
  (lambda (n 1)
    (cond
      ((eq n 0) nil)
      (t (cons (car 1) (firstn (- n 1) (cdr 1)))))))

(defun build
  (lambda (x y)
    (cond
      ((eq x nil) nil)
      (t (cons (car x) y) (build (cdr x) y))))))

(defun do_newS
  (lambda (m n)
    (setq idle_temp (max clockTicks (untag_stack (+ (+ m n) 1)))))
    (cond ((eq idle_temp clockTicks) (setq overhead (+ overhead

(setq overhead (setq idle_temp (max clockTicks (untag_stack (+ (+ m n) 1))))
    (cond ((eq idle_temp clockTicks) (setq overhead (+ overhead

(setq stack (cons (stk1) (stk2r)))))

(defun do_S
  (lambda ()
    (check_args 3 'Sctr)
    (my_trace (list 'do_S))
    (setq stack
      (cons (stk1) (cons (stk3) (cons (stk2) (stk3r) (stk3r))))))

(defun do_B
  (lambda ()
    (check_args 3 'Bctr)
    (my_trace (list 'do_B))
    (setq stack (cons (stk1) (cons (stk2) (stk3r) (stk3r))))))

(defun do_C
  (lambda ()
    (check_args 3 'Cctr)
    (my_trace (list 'do_C))
    (setq stack (cons (stk1) (cons (stk3r) (stk3r) (stk3r))))))

(defun toss
  (lambda (n 1)
    (cond
      ((eq n 0) 1)
      (t (toss (- n 1) (cdr 1))))))

(defun find
  (lambda (n 1)
    (cond
      ((eq n 0) nil)
      ((eq n 1) (car 1))
      (t (find (- n 1) (cdr 1))))))

(defun extract
  (lambda (s n 1)
    (firstn n (toss (- s 1) 1))))

(defun firstn
  (lambda (n 1)
    (cond
      ((eq n 0) nil)
      (t (cons (car 1) (firstn (- n 1) (cdr 1)))))))

(defun build
  (lambda (x y)
    (cond
      ((eq x nil) nil)
      (t (cons (car x) y) (build (cdr x) y))))))

(defun do_newS
  (lambda (m n)
    (setq idle_temp (max clockTicks (untag_stack (+ (+ m n) 1))))
    (cond ((eq idle_temp clockTicks) (setq overhead (+ overhead

(def do_X
  (lambda (m n k)
    (setq idle_temp (max clock_ticks (untag_stack (+ (+ m n) 1))))
    (cond ((neq idle_temp clock_ticks) (setq overhead (+ overhead
                  (min local_overhead (- idle_temp clock_ticks))))))
    (setq idle_time (+ idle_time (- idle_temp clock_ticks)))
    (cond ((neq idle_temp clock_ticks) (my_trace (list
                  'wait (- idle_temp clock_ticks))))
      (setq clock_ticks idle_temp)
    (cond ((> (hole_count 1 (+ (+ m n) 1)) 0)
      (my_push (list 'do_newS m n))
      (my_push resumeflag)
      (my_top))
    (t
      (cond ((< (length stack) (+ (+ m 1) n))
        (my_push (list 'do_newS m n))
        (my_push resumeflag)
        (join))
      (check_args (+ (+ m 1) n) 'newS)
      (my_trace (list 'do_newS m n))
      (setq stack
        (append
          (build (extract 1 (+ m 1) stack)
            (toss (+ m 1))
            (extract 1 (+ (+ m 1) n) stack))
          (toss (+ (+ 1 m) n) stack)))
      (startprocs 2 (+ m 1))
      (update_clocks)
      (cond (resumed (setq resumed nil) (do_top)))))))

(def do_Y
  (lambda (m n k)
    (setq idle_temp (max clock_ticks (untag_stack (+ (+ m n) 1))))
    (cond ((neq idle_temp clock_ticks) (setq overhead (+ overhead
      (min local_overhead (- idle_temp clock_ticks))))))
    (setq idle_time (+ idle_time (- idle_temp clock_ticks)))))
  (cond ((neq idle_temp clock_ticks) (my_trace (list
    'wait (- idle_temp clock_ticks))))))
  (setq clock_ticks idle_temp)
  (cond ((>(hole_count (+ (+ m n) 1)) 0)
    (my_push (list 'do_Y m n k))
    (my_push resumeflag)
    (my_top))
  (t
    (cond ((<= (length stack) (+ (+ m 1) n))
      (my_push (list 'do_Y m n k))
      (my_push resumeflag)
      (my_top))
    (check_args (+ (+ m) 1) 'Yctr)
    (my_trace (list 'do_Y m n k))
    (setq stack
      (append (append (build (extract 1 k stack)
        (toss (+ m 1))
        (extract (+ (+ m 1) n) stack))
      )
      (extract (+ 1 m) (+ (- m k) 1) stack)
      )
      (toss (+ (+ 1 m) n) stack)))))
  (startprocs 2 (+ m 1))
  (update_clocks)
  (cond (resumed (setq resumed nil) (do_top)))
))
(def do_Z
  (lambda (m n k)
    (setq idle_temp (max clock_ticks (untag_stack (+ (+ m n) 1))))
    (cond ((neq idle_temp clock_ticks) (setq overhead (+ overhead
      (min local_overhead (- idle_temp clock_ticks))))))
    (setq idle_time (+ idle_time (- idle_temp clock_ticks)))))
  (cond ((neq idle_temp clock_ticks) (my_trace (list
    'wait (- idle_temp clock_ticks))))))
  (setq clock_ticks idle_temp)
  (cond ((>(hole_count (+ (+ m n) 1)) 0)
    (my_push (list 'do_Z m n k))
    (my_push resumeflag)
    (my_top))
  (t
    (cond ((<= (length stack) (+ (+ m 1) n))
      (my_push (list 'do_Z m n k))
    )
  )
(my_push resumeflag)
  (join))
(check_args (+ (+ m 1) n) 'Zctr)
(my_trace (list 'do_Z m n k))
(setq stack
  (append (append (extract 1 k stack)
     (append (build (list
       (find (+ k 1) stack))
      (toss (+ m 1)
       (extract 1 (+ (+ m 1) n)
       stack)))
     (extract (+ k 2) (- m k) stack)
     )
  (toss (+ (+ 1 m) n) stack)))
(cond ((eq k 0) (startprocs 2 (+ m 1))
  (t (startprocs (+ k 1) (+ m 1))))
(update_clocks)
(cond (resumed (setq resumed nil) (do_top)))))

(def do_newI
  (lambda (m n)
    (setq idle_temp (max clock_ticks (untag_stack (+ (+ m n) 1))))
    (cond ((neq idle_temp clock_ticks) (setq overhead (+ overhead
      (min local_overhead (- idle_temp clock_ticks)))))
    (setq idle_time (+ idle_time (- idle_temp clock_ticks)))
    (cond ((neq idle_temp clock_ticks) (my_trace (list
      'wait (- idle_temp clock_ticks)))))
    (setq clock_ticks idle_temp)
    (cond ((> (hole_count 1 n) 0 )
      (my_push (list 'do_newI m n))
      (my_push resumeflag)
      (my_top))
    (t (cond ((< (length stack) n)
      (my_push (list 'do_newI m n))
      (my_push resumeflag)
      (join))
    (check_args n 'newlctr)
    (my_trace (list 'do_newI m n))
    (setq stack
      (append (list (find m stack))
      (toss n stack)))
    (cond (resumed (setq resumed nil) (do_top)))))

(def do_newK
  (lambda (n)
    (setq idle_temp (max clock_ticks (untag_stack (+ (+ m n) 1))))
    (cond ((neq idle_temp clock_ticks) (setq overhead (+ overhead
      (min local_overhead (- idle_temp clock_ticks)))))
    (setq idle_time (+ idle_time (- idle_temp clock_ticks)))
    (cond ((neq idle_temp clock_ticks) (my_trace (list
      'wait (- idle_temp clock_ticks)))))

(setq clock_ticks idle_temp)
(cond (( > (hole_count 1 (+ n 1)) 0 )
  (my_push (list 'do_newK n))
  (my_push resumeflag)
  (my_top))
(t
  (cond (( < (length stack) (+ 1 n))
    (my_push (list 'do_newK n))
    (my_push resumeflag)
    (join)))
  (check_args (+ 1 n) 'newKctr)
  (my_trace (list 'do_newK n))
  (setq stack
    (append (list stk1))
    (toss (+ n 1) stack)))
  (cond(resumed(setq resumed nil) (do_top)))))))

(def init
  (lambda ()
    (setq Xctr 0 newSctr 0 Yctr 0 Zctr 0 newlctr 0 newKctr 0)
    (setq stack '(a b c d e f l m n o p w x y z)))
(def new_top
  (lambda ()
    (setq newtopctr (+ newtopctr 1))
    (setq idle_temp (max clock_ticks (untag_stack 1)))
    (cond ((neq idle_temp clock_ticks) (setq overhead (+ overhead
        (min local_overhead (- idle_temp clock_ticks)))))
      (setq idle_time (+ idle_time (- idle_temp clock_ticks)))
      (cond ((neq idle_temp clock_ticks) (my_trace (list
        wait (- idle_temp clock_ticks))))
        (setq clock_ticks idle_temp)
        (cond ((eq (stkl) resumeflag) (setq local (stk1))
          (setq stack (stk2r))
          (setq resumed 't)
          (eval local)))
        (cond ((numberp (stkl)) (go (no_quote (retstk1))))
          ((atom (stkl))
            (progn (setq tem (stkl))
              (setq stack (stk1r))
              (cond ((atom stack)
                (cond ((not (null stack))
                  (setq stack (list stack))))
                (go (no_quote tem)))))
            (t
              (progn (setq oldpair (stkl))
                (setq stack (cons (caar stack)
                  (ourappend (cdr stack) (stk1r))))
                (new_top))))))
  (def ourappend
    (lambda (x y)
      (cond
((> i (length stack)) nil)
((is_hole (nthelem i stack)) 't)
((numberp (nthelem i stack)) nil)
((<= avail_procs procs_jn_use) nil)
(t (setq stack (append (firstn (-il) stack)
    (append (list (fork (nthelem i stack) nil))
    (nthcdr i stack)))) 't)
)
)
)
(def startprocs
  (lambda (i n)
    (cond
      ((eq i n) (cond ((start_arg i) 1) (t 0)))
      (t (+ (cond ((start_arg i) 1) (t 0))
        (startprocs (+ i 1) n)))
    )))
(def update_clocks
  (lambda ()
    (setq master_clock (+ master_clock clock_ticks))
    (setq clock_ticks 0)
  ))
(def hole_count
  (lambda (i n)
    (cond
      ((eq i n) (cond ((is_hole (nthelem i stack)) 1) (t 0)))
      (t (+ (cond ((is_hole (nthelem i stack)) 1) (t 0))
        (hole_count (+ i 1) n)))
    )))
(def test_for_holes
  (lambda (n)
    (cond ((> (hole_count 1 n) 0) 't)
      (t nil)))
)
(def is_hole
  (lambda (x)
    (def look (lambda (y)
            (cond ((null y) nil) ((eq x (car y)) 't)
               (t (look (cdr y)))))))
    (look hole_ids)))
(def appendl (lambda (x y) (append x (list y))))
(def fork
  (lambda (mystack myret_stack)
    (setq procs_in_use (+ procs_in_use 1))
    (cond ((< max_in_use procs_in_use) (setq max_in_use procs_in_use))
      (setq forkctr (+ forkctr 1))
      (cond ((atom mystack) (setq mystack (list mystack)))
        (cond ((atom (cdr mystack)) (cond ((not (null (cdr mystack)))
          (setq mystack (list (car mystack) (cdr mystack))))))
        (cond ((atom myret_stack) (setq myret_stack (list myret_stack)))
          (cond ((atom (cdr myret_stack)) (cond ((not (null (cdr myret_stack)))
            (setq myret_stack (list (car myret_stack) (cdr myret_stack))))))
          (setq ready (appendl ready mystack))
          (setq hole_ids (appendl hole_ids (gensym))))
      )
  ))
(setq ready 1 (appendl ready 1 myret_stack))
(setq timers (appendl timers 0))
(setq master_clocks (appendl master_clocks 0))
(setq trace_stuff (appendl trace_stuff (list mystack)))
(setq idle_list (appendl idle_list 0))
(setq overhead_times (appendl overhead_times overhead_constant))
(setq overhead_in (appendl overhead_in (list (our_length mystack))
                      (depth mystack))))
(rplaca (last current_trace) (appendl (car (last current_trace))
                      (last hole_ids)))

(car (last hole_ids))
)
(def join
(lambda ()
  (setq procs_in_use (- procs_in_use 1))
  (update_clocks)
  (setq master_clock (+ master_clock overhead_constant))
  (setq proc_switches (+ proc_switches 1))
  (setq current_trace (appendl current_trace stack))
  (setq joinctr (+ joinctr 1))
  (setq ready (cdr ready))
  (setq over_out (our_length stack))
  (setq over_out1 (depth stack))
  (cond ((eq (length stack) 1) (setq stack (car stack))))
  (dsubst (list 'timeflag (list master_clock overhead) stack)
           (car hole_ids) ready)
  (dsubst (list 'timeflag (list master_clock overhead) stack)
           (car hole_ids) ready)
  (setq time_counts (cons (list (car hole_ids) master_clock)
                           time_counts))
  (setq traces (cons (list (car hole_ids) current_trace)
                     traces))
  (setq idle_summaries (cons (list (car hole_ids) idle_time)
                               idle_summaries))
  (setq overhead_summaries (cons (list (car hole_ids) overhead)
                                overhead_summaries))
  (setq timers (cdr timers))
  (setq master_clocks (cdr master_clocks))
  (setq trace_stuff (cdr trace_stuff))
  (setq idle_list (cdr idle_list))
  (setq overhead_times (cdr overhead_times))
  (setq hole_ids (cdr hole_ids))
  (setq ready 1 (cdr ready))
  (setq stack (car ready))
  (setq ret_stack (car ready))
  (setq clock_ticks (car timers))
  (setq master_clock (car master_clocks))
  (setq current_trace (car trace_stuff))
  (setq idle_time (car idle_list))
  (setq overhead (car overhead_times))
  (setq overhead_in (cdr overhead_in))
  (new_top)))
(def my_rotate
  (lambda (x y)
    (set x (appendl (cdr (eval x)) (eval y)))
    (set y (car (eval x))))
)

(def do_top
  (lambda ()
    (check_args 0 'topcnt)
    (my_trace (list 'do_top))
    (cond ((> (length ready) 1) (setq proc_switches
        (+ proc_switches 1)))

      (setq ready (appendl (cdr ready) stack))
      (setq stack (car ready))
      (setq ready1 (appendl (cdr ready1) ret_stack))
      (setq ret_stack (car ready1))
      (my_rotate 'timers 'clock_ticks)
      (my_rotate 'master_clocks 'master_clock)
      (my_rotate 'trace_stuff 'current_trace)
      (my_rotate 'idle_list 'idle_time)
      (my_rotate 'overhead_times 'overhead)
      (setq hole_ids (appendl (cdr hole_ids) (car hole_ids)))
      (setq overhead_in (appendl (cdr overhead_in) (car overhead_in)))
      (new_top)
    )))

(def my_top
  (lambda ()
    (setq clock_ticks (- clock_ticks 1))
    (do_top)
  ))

(def our_init (lambda (x)
    (setq resumed nil)
    (setq resumeflag 'spflag)
    (setq top_temp Ldeb)
    (setq ready (list nil)) ; processor stack frames
    (setq ready1 (list nil)) ; processor return stack frames
    (setq hole_ids (list (gensym))) ; return hole ids
    (setq timers (list 0)) ; processor clock_ticks
    (setq master_clocks (list 0)) ; processor master_clock ticks
    (setq trace_stuff nil) ; process traces
    (setq idle_list (list 0)) ; processor idle times
    (setq overhead_times (list 0)) ; overhead times
    (setq idle_summaries nil) ; idle_time summaries
    (setq trace_summaries nil) ; overall trace summaries
    (setq overhead_in (list 0)) ; input expense
    (setq overhead_summaries nil) ; the overhead times
    (setq stack x))

(def myprint (lambda ()
    (print 'stack=) (print stack) (terpri)
    (print 'ready=) (print ready) (terpri)
    (print 'ready1=) (print ready1) (terpri)
    (print 'hole_ids=) (print hole_ids) (terpri)
  ))
(def call1
  (lambda (cont)
    (cond
      ((<= avail_procs (+ procs_in_use 1)) (old_call1 cont))
      (t (my_push cont)
          (my_trace (list 'call1 cont))
          (my_top))))))

(def call2
  (lambda (cont)
    (cond
      (eq (stk2) 'oldcall) (setq stack (list (stk1))) (old_call2 cont))
      (t
        (my_push cont)
        (my_trace (list 'call2 cont))
        (my_top))))

(defun old_do_plus
  (lambda ()
    (my_trace (list 'old_do_plus))
    (setq tem (retstk2))
    (result (+ tem (stk1)) 'plusctr)))

(defun do_plus
  (lambda ()
    (cond ((eq (stk2) 'oldcall) (setq stack (list (stk1))) (old_do_plus))
      (t
        (check_only 2)
        (setq binopctr (+ binopctr 1))
        (fix_args top_temp)
        (check_args 2 'plusctr)
        (my_trace (list 'do_plus))
        (setq idle_temp (max clock_ticks (untag_stack 2)))
        (cond ((neq idle_temp clock_ticks) (setq overhead (+ overhead
            (min local_overhead (- idle_temp clock_ticks)))))
            (setq idle_time (+ idle_time (- idle_temp clock_ticks)))
            (cond ((neq idle_temp clock_ticks) (my_trace (list
                'wait (- idle_temp clock_ticks))))
                (setq clock_ticks idle_temp)
                (update_clocks)
                (setq stack (cons (+ (stkl) (stk2)) (stk2r))))))
    ))

(defun old_do_times
  (lambda ()
    (my_trace (list 'old_do_times))
    (setq tem (retstk2))
    (result (* tem (stk1)) 'timesctr)))

(defun do_times
  (lambda ()
    (cond ((eq (stk2) 'oldcall) (setq stack (list (stk1)))
          (old_do_times))
      (t
        (check_only 2)
        (setq binopctr (+ binopctr 1))
        (fix_args top_temp)
        (check_args 2 'timesctr)
(my_trace (list 'do_times))
(setq idle_temp (max clock_ticks (untag_stack 2)))
(cond ((neq idle_temp clock_ticks) (setq overhead (+ overhead
(min local_overhead (- idle_temp clock_ticks))))))
(setq idle_time (+ idle_time (- idle_temp clock_ticks)))
(cond ((neq idle_temp clock_ticks) (my_trace (list
'wait (- idle_temp clock_ticks)))))))
(setq idle_time (+ idle_time (- idle_temp clock_ticks)))
(cond ((neq idle_temp clock_ticks) (my_trace (list
"wait (- idle_temp clock_ticks))")
(setq clock_ticks idle_temp)
(update_clocks)
(setq stack (cons (* (stkl) (stk2)) (stk2r)))))))

(defun old_do_minus
  (lambda ()
    (my_trace (list 'old_do_minus))
    (setq tem (retstk2))
    (result (- tem (stkl)) 'minusctr))
(defun do_minus
  (lambda ()
    (cond ((eq (stk2) 'oldcall) (setq stack (list (stkl)))) (old_do_minus))
    (t
      (check_only 2)
      (setq binopctr (+ binopctr 1))
      (fix_args top_temp)
      (check_args 2 'minusctr)
      (my_trace (list 'do_minus))
      (setq idle_temp (max clock_ticks (untag_stack 2)))
      (cond ((neq idle_temp clock_ticks) (setq overhead (+ overhead
(min local_overhead (- idle_temp clock_ticks))))))
      (setq idle_time (+ idle_time (- idle_temp clock_ticks)))
      (cond ((neq idle_temp clock_ticks) (my_trace (list
'wait (- idle_temp clock_ticks)))))))
      (setq clock_ticks idle_temp)
      (update_clocks)
      (setq stack (cons (- (stkl) (stk2)) (stk2r))))))
(def old_do_eq
  (lambda ()
    (my_trace (list 'old_do_eq))
    (setq tem (retstk2))
    (result (equal tem (stkl)) 'eqctr))
(defun do_eq
  (lambda ()
    (cond ((eq (stk2) 'oldcall) (setq stack (list (stkl)))) (old_do_eq))
    (t
      (check_only 2)
      (setq binopctr (+ binopctr 1))
      (fix_args top_temp)
      (check_args 2 'eqctr)
      (my_trace (list 'do_eq))
      (setq idle_temp (max clock_ticks (untag_stack 2)))
      (cond ((neq idle_temp clock_ticks) (setq overhead (+ overhead
(min local_overhead (- idle_temp clock_ticks))))))
      (setq idle_time (+ idle_time (- idle_temp clock_ticks))))))
(cond ((neq idle_temp clock_ticks) (my_trace (list 'wait (- idle_temp clock_ticks)))))
(setq clock_ticks idle_temp)
(update_clocks)
(setq stack (cons (equal (stk1) (stk2)) (stk2r))))
)
(def untag_stack
  (lambda (n)
    (prog (local newlist i)
      (cond ((> n (length stack)) (setq n (length stack)))
        (setq newlist nil)
        (setq local 0)
        (setq local_overhead 0)
        (setq i 1)
        Loclab (setq newelem (remove_tag (nthelem i stack)))
        (setq newlist (append newlist (list newelem)))
        (cond ((< i n) (setq i (+ i 1))
          (go Loclab))
          (t (setq stack (append newlist (nthcdr i stack)))
            (return local))))
    )))
(def remove_tag
  (lambda (x)
    (cond ((atom x) x)
      (t (cond ((eq (car x) 'timeflag)
        (setq local (max local (caadr x)))
        (setq local_overhead (max local_overhead (cadadr x)))
        (setq x (caddr x))
      )
      (t x)))
    )))
(def set_overhead_constant
  (lambda (x) (setq overhead_constant x)))
(def restart
  (lambda ()
    (start)
    (our_init nil)))