COMBINATOR REDUCTIONS ON A LISP MACHINE

by

NİLGÜN MAT

A THESIS

Presented to the Department of Computer
and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
Master of Science

June 1985

APPROVED: ______________________
Dr. John S. Conery

An Abstract of the Thesis of

Nilgün Mat for the degree of Master of Science

in the Department of Computer and Information Science

to be taken June 1985

Title: COMBINATOR REDUCTIONS ON A LISP MACHINE

Approved: ______________________________

Dr. John S. Conery

A number of techniques exist for implementing applicative languages. One of the techniques, based on Combinatory Logic, implements applicative languages as combinator reductions. This thesis describes an implementation on the Symbolics 3600 Lisp machine of normal order reductions with either copied or shared values for a subset of SASL using Turner's combinator technique. These two implementations are compared with respect to the time and memory requirements of reductions. Also, the efficiency of reductions with and without the optimisation rules are investigated. Finally, the user is provided with the option of displaying the underlying structure and the transformations on copied and shared reductions.

VITA

NAME OF AUTHOR: Nilgün Mat

PLACE OF BIRTH: Istanbul, Turkey

DATE OF BIRTH: March 24, 1960

UNDERGRADUATE AND GRADUATE SCHOOLS ATTENDED:

University of Reading, Britain
University of Oregon

DEGREES AWARDED:

Bachelor of Science, 1982, University of Reading, Britain
Master of Science, 1985, University of Oregon

AREAS OF SPECIAL INTEREST:

Programming Languages

PROFESSIONAL EXPERIENCE:

Teaching Assistant, Department of Computer and Information Science, University of Oregon, Eugene, 1983-85

Student Programmer, Computing Center, University of Oregon, Eugene, 1983-84

ACKNOWLEDGEMENTS

The author wishes to express her appreciation to her thesis adviser John Conery for his guidance and support throughout the study. In addition special thanks are due to Lajos Ronjai and Steve Fickas for their helpful comments on this paper. The author would also like to extend her sincere thanks to Professor David Turner for replying to her letter so quickly when she needed help.

Anneme ve Babama

or

To my parents

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

In the last decade, applicative languages (also known as functional programming languages) have been the center of considerable interest and research. The reason for such interest stems from the clean semantics and the elegant programming style associated with these languages. The acceptance of applicative languages, however, has suffered from their inefficent implementations in comparison to the more efficent implementations of the imperative languages. Nevertheless, applicative languages have recently become more attractive due to the discovery of improved implementation methods.

This thesis uses Turner's combinator reduction technique (Turner, 1979) to implement a subset of his SASL language on the Symbolics 3600 Lisp Machine. The first two sections of this chapter feature the characteristics and implementation methods of applicative languages. The last two sections of the chapter give the description and the motivation of the thesis.

Power of Applicative Languages

In an applicative language, the central mechanism is function application. The notion of an applicative language is best illustrated through Richards' description of applicative:

> ...a term which refers to notation in which computation consists solely of applications of functions to arguments. Imperative operations such as assignment statements and input and output commands are entirely absent from applicative notations. Also absent are the concepts of machine state...(Richards, 1982:4)

Applicative languages allow a higher level description of a solution, which raises the level of programming and decreases chances of error. For example, a function that adds one to every element of a list has much in common with a function that tests for zeroes in a list. The only difference is the particular function that is applied to list elements. Thus a desirable commodity is the ability to abstract out the common patterns and write a function which takes as its argument another function (such as "add1" or "zerop") and applies it to every element of a list. Abstracting out a recurring pattern frees the user to concentrate on the essence of the problem.

Models of computation for applicative languages are based on lambda-calculus and Church's combinators. This results in clean semantics with side-effect free operations. The implication is that functions can be combined in ways which can supress details of loop control. As an example, consider the "map" function which takes a function and a list as arguments, and applies the function to each element in the list. In an applicative style "map" can be written as:

```
map f [] = []
map f(a:x) = f a : map f x
```

Throughout the thesis the notation that f x will be used to indicate f applied to x. Thus given an empty list, [], "map" returns an empty list. If the list is non-empty then "map" applies the function "f", which is passed as a parameter, to the first element in the list and recursively calls itself for the rest of the elements in the list.

In an imperative language "map" would correspond to

```
procedure map (f,x,y);
 integer i;
 for i:= 1 to n do
  y[i] := f (x[i]);
```

In comparing the two programs, the following is noticed:

(1) the former is completely general whereas the latter requires _n_, size of the list, as part of the code.

(2) the applicative code allows ease in combining functions as in application of "map" to function "f". Similarly, a function "g" can be applied to the readily

defined "map" function, i.e g map f x. In most imperative languages, the function "f" usually can not be passed as a parameter. In a few imperative languages (such as Algol-60) which allow function parameters, the process is awkward due to side-effects.

(3) the applicative program operates on conceptual units versus assignment statement at a time computing.

(4) the first code has no states, only values; in the latter code statements operate on a state.

Another nice property of applicative languages is the availability of infinite data objects. Consider a program that finds the first integer which has a certain property P. There is a need for two different processes. One process will generate integers while another process will test the integers for a property P. The integers starting from n can be generated as follows:

```
ints n = n :  ints n + 1
```

A function which finds the first integer for which a function f is true can be described as:

```
first f (x:y) = f (x) -> x;
                first f y
```

If an integer i is true for f i, then first f i returns the integer, otherwise, next integer is tested for the property. Let prop x be a function that returns true if integer x has property P. By combining the processes of integer generation and property testing our program is

```
first prop (ints 1)
```

This program will generate as many integers as required to find the first integer which satisfies the test condition. Note that "ints" defines an infinite list of integer, but the interpreter generates only as many of these integers as are required by the process named "first."

Another feature of applicative languages is that they support non-strict functions, i.e. functions capable of returning a result even when one of the parameters is undefined. For example, consider the expression:

```
x = 0 -> 1; y
```

The above conditional, when x is zero, has the value 1, otherwise the value y. Such an expression should be non-strict, since the value should always be 1 when x is zero, whether or not the value of y exists.

Finally, the clean semantics and higher-order functions lead into possibility of exploiting the inherent parallelism in a problem. For example, an imperative program to compute the product of two vectors u and v of size 10 elements uses a loop to multiply each corresponding element, assigning the product to another vector uv:

```
for i:= 1 to 10 do
 uv[i] := u[i] * v[i];
```

The order of multiplication, i.e. multiplication of u[1] by v[1] first, then u[2] by v[2], and so on is not part of the problem. Multiplying u and v in parallel, i.e. performing elementwise multiplications simultaneously, is usually not possible in an imperative interpreter. Due to referential transparency, interpreters for applicative languages are free to use any order. An example is the unraveling interpreter for Id (Arvind et al,1978).

Implementation Considerations in Applicative Languages

A main issue in language implementation is keeping track of variable bindings. Variables include formal parameters like n in this definition of factorial:

```
fac n =  n = 0 -> 1; n * fac (n - 1)
```

Another example is the local variable x in the expression:

```
(x + 1) * (x - 3) where x = 7
```

Each variable has a scope, and the variable can be replaced by its value within this scope without altering its meaning. In order to achieve such replacement, the variable and its value need to be kept together.

Languages have different methods for associating a variable and its value. Imperative languages with static scoping such as Pascal use activation record stacks with displays to keep track of their identifiers during execution of procedures and block structures. Languages like LISP use association lists (a-lists) to keep track of variable bindings. Most applicative language implementations use a-lists in the form of a name-value pair. A number of implementations such as the interpreter of Henderson and Morris use closures, i.e. name-environment pairs to evaluate variable bindings. Implementations based on Church's combinators, such as the interpreters of Johnsson's and Turner, compile away references to variables so that the compiled code has no variables names, and the run-time system does not have to keep track of variable bindings.

Description of the Thesis

The thesis work is based on Turner's implementation technique for an applicative language called "St. Andrews Static Language" (SASL) (Turner,1982). This thesis describes the implementation of a language referred to as

PSASL (for prefix SASL). The PSASL interpreter is implemented in ZetaLisp on a Symbolics 3600 Lisp machine.

PSASL is a subset of SASL with some syntactic changes in the notation. The PSASL interpreter implements two different reduction engines using a technique described by Turner (1984). Turner's technique, known as graph reduction, is an alternative to the more common method of string reduction. Generally, string reductions are less efficient than graph reductions, in which portions of the computation can be shared. However, no figures are available as to how much worse the former is when executing a typical program.

Motivation for the Thesis

This thesis was motivated by:

(1) The interest to understand Turner's implementation technique, especially to understand how it optimises execution of recursive functions.

(2) The aim to quantify the relative efficiencies of graph reductions with string reductions.

(3) A chance to investigate the clarity brought by displaying the combinator reductions to the user. A disadvantage with a language like SASL is that it is hard to tell users why a program failed. Visual display is an attempt to solve this problem.

CHAPTER II

IMPLEMENTATION TECHNIQUES FOR APPLICATIVE LANGUAGES

A number of different techniques exist for implementing applicative languages on sequential machines. Most implementations of applicative languages evaluate arguments to functions either before (applicative order) or after (normal order) function application. The choice of evaluation strategy is important: if a program has a value, a normal order interpreter will terminate with that value, whereas in applicative order evaluation no such termination is guaranteed. For example, consider the function "c7" which produces the number seven regardless of its argument.

```
c7 f y = 7
   where f y = y + y
```

"c7" may or may not terminate depending on the evaluation strategy used. The normal order evaluation will return 7 since the arguments f and y are not needed in the function evaluation. The applicative order evaluation will evaluate f(y) first, and "c7" will not terminate if evaluation of f(y) never terminates. Implementation of infinite data objects also requires normal order evaluation, so that only the needed arguments will be evaluated leading to the termination of the program.

A major drawback of normal order evaluation, however, is its inefficiency. A concern in normal order evaluation is the multiple evaluations of a common subexpression in a given expression, i.e. there is no "sharing of values". For example in

```
(2 * x * y) + (7 * y)
                where x = y - 2
                      y = 1 + 3
```

The answer is 44. This result can be reached in two ways:

(1) by evaluating arguments first, calculating the value of y and x to be 4 and 2, and then subsituting the numbers appropriately in the evaluation the expression (applicative order).

(2) by subsituting the expressions for y and x in the main expression and performing the calculations (normal order). In other words, evaluate

(2 * ((1+3) - 2) * (1+3)) + (7 * (1+3)).

In the second method, calculation of y is done three times, as opposed to the single calculation performed in the first case. Previous implementations of applicative languages have used one or the other of these orders. Normal order interpreters are also known as lazy or delayed evaluation interpreters.

The SECD machine of Landin and the Lisp interpreter of McCarthy (1960) use applicative order in evaluating arguments to functions. Henderson and Morris (1976), Wadsworth (1971) and Johnsson (1983) use delayed evaluation of arguments; however, each differ in their implementation.

Henderson and Morris's lazy interpreter keeps the environment as a name-value association list and evaluates a common subexpression more than once. Wadsworth and Johnsson maintain a graph structure for the applicative expression where copying of some common subexpressions is avoided through its graph structure. This results in fewer reductions; but, for other expressions where copies are needed, the copying procedure becomes costly since questions arise as to what and how much of the structure should be copied. However, the method of detecting common subexpressions during reductions in both implementations are shown by Arvind to be equivalent (Arvind et al, 1982).

A better method for finding common subexpressions is detection of these expressions during the compilation of an expression rather than its reduction. Johnsson's G-machine uses Hughes' method to detect common subexpressions during compilation (Hughes, 1982). Unfortunately, all the methods described involve some overhead. Hughes' algorithm depends on the order of compilation. Henderson and Morris has a major overhead due to the construction of environments.

Wadsworth's interpreter requires a search through the graph to detect bound variables.

Combinator Machines

The inefficiency seen in normal order reductions in some applicative languages is due to their lambda style reducer implementations. Normal order evaluation for a lambda reducer involves forming a closure for the argument of the function in order to postpone its evaluation. This process is potentially expensive and in practice most lambda reducers implement applicative order evaluation (Jones, 1982).

An alternative implementation, based on combinatory logic, eliminates all variables from the object program by a process known as bracket abstraction, introducing constants called combinators (defined below). A compilation algorithm produces the combinatory code, which then is executed on a graph reduction machine. Combinators lend themselves naturally to normal order evaluation. Also, they provide a good basis for program transformation and verification.

Johnsson's G-machine provides an efficient implementation of Hughes' abstraction algorithm and his super-combinator approach. In the G-machine the set of combinators change for a given expression with the notion that the best set of combinators are used for a particular expression. Turner's abstraction algorithm is not as efficient, in that repeated passes are needed over the combinator expression, but, it uses a fixed set of simple combinators. Turner's machine, due to its simplicity, is very attractive as it lends itself to hardware implementation.

Today, there are implementations of combinator reducers in hardware. Two of these machines are built in United Kingdom: CRS/1 with a lambda-to-combinator converter (Beale, 1982) and SKIM designed by Cambridge University. The Burroughs Research Center at Austin, Texas is also in the process of building a combinator reduction machine, NORMA, based on Turner's implementation method.

Turner's Combinator Approach

Combinators perform the same tasks as lambda-calculus operations, but without the use of variable bindings. Thus the inefficiencies due to management of variable bindings in applicative language implementations based on lambda-calculus are not present in combinatory logic.

The notion of a combinator can be best illustrated through an example. In arithmetic, the commutative law of addition can be expressed as

for all x,y: $x + y = y + x$

This law can be expressed without the variable bindings by defining a function A

$$A\ x\ y = x + y \qquad \{\text{for all } x,\ y\}$$

and by introducing an operator C̲ which transforms sentences about functions. We define C̲ to operate on a function f and objects x and y as follows:

$$\underline{C}\ f\ x\ y = f\ y\ x \qquad \{\text{for all } f,x,y\}$$

Then the commutative law can be expressed simply as:

$$\underline{C}\ A = A$$

The operator C̲ is called a combinator (Hindley et.all, 1972).

Turner, in his implementation of a combinator reduction machine, defines an applicative language called SASL (St. Andrews Static Language). This language, although simple and small, has all the characteristics of other applicative languages (Turner, 1984).

Turner's implementation evaluates a SASL expression in the following way:

(1) The SASL expression is converted to a combinator expression via the abstraction algorithm using the fixed set of combinators S, K, I, C, B, Y (these will be defined below).

(2) The combinator expression is evaulated by a reduction machine until the result of the SASL expression is obtained.

In the first part of the implementation an expression such as

```
fac 4
  where
    fac n = n = 0 -> 1; n * fac (n - 1)
```

is compiled to its equivalent combinator expression:

```
C I 4(Y (B(S(C(B cond(eq 0)) 1))
                (B(S times)(C minus 1)))))))
```

The upper case letters are combinators, and the remaining symbols are function names and constant objects. Notice that all variables have been removed.

In the second part of the implementation, the compiled code, in the form of a tree structure, is passed to the reduction machine. The reduction machine successively reduces the tree into a final output value. During execution, subtrees representing common subexpression become shared, and the tree is tranformed to a graph structure. So "fac 4" combinator code is progressively reduced until the result 24 is obtained.

Turner's technique of normal graph reductions with shared graphs has the advantage of combining termination properties of normal order reductions with the efficiency of applicative order reductions. The abstraction algorithm, definitions of combinators, and execution algorithm are given in the next chapter.

CHAPTER III

IMPLEMENTATION OF NORMAL REDUCTIONS

This chapter describes an implementation of bracket abstraction and reduction algorithms on the Symbolics 3600 Lisp machine in Zetalisp running under release 5.1. The input is a subset of SASL called PSASL, for "prefix SASL", in which every function application is written in prefix form. So, for example, the user would write

```
times (plus x 3) (minus x 4) where x = 22
```

instead of the SASL expression

```
(x + 3) * (x + 4) where x = 22
```

The output is the result of the reduction. The reduction engine includes a facility to display the underlying data structure of the input PSASL expression and the transformations on this expression.

Implemented Set of PSASL

A very small subset of SASL is implemented. The justification for such a small set is that the aim was not to implement a SASL interpreter but to understand and compare normal order string and graph reductions. Appendix includes a brief description of the PSASL grammar rules in BNF notation. A PSASL program consists of:

Objects: Two types of objects are available, numbers (integers) and truth values (true, false).

Primitives: The primitive operations include plus, minus, times, div, cond, eq, less, grt, lesseq, grteq, noteq, and, or.

Lexical conventions: The input PSASL expression is different than the SASL expression as mentioned above. The former is in prefix monadic notation while the latter has an infix notation. For example, a SASL expression to calculate the third Fibonacci number corresponds to

```
fib 3 where fib x = x = 0 -> 1 ;
                    x = 1 -> 1 ; fib (x - 1) + fib (x - 2)
```

In PSASL this is written

```
fib 3 where
      fib x = cond (eq 0 x) 1
              (cond (eq 1 x)
              (plus (fib (minus x 1)) (fib (minus x 2))))
```

Another restriction in the implementation is in use of where clauses. In Turner's SASL language there are two types of where clauses

(1) nested

(2) multiple

An example of a nested where clause is the expression

```
(3 * y where y = z + 10) where z = 5
```

An example of a multiple where clause on the other hand is

```
x * (y + x) where y = 3
                  x = 7
```

in which the where clasuse defines two or more values at the same level. The above expressions result in 45 and 70 respectively. In this implementation nested wheres (like the first example) can be handled. However, multiple wheres are not included.

Abstraction Algorithm

PSASL expressions are compiled with Turner's abstraction algorithm (Turner, 1984). The abstraction algorithm converts the input PSASL expression into a constant form by abstracting the variables and introducing the combinators.

A PSASL expression "el" applied to another expression "e2" is represented as the juxtaposition of two expression, i.e. "el e2", and the function application is left associative. This example shows the result of f(x) being applied to g(y).

    f x (g y)

In the above expression "f" is applied to "x" and the resulting function is applied to "g y". The parenthesis in "g y" show that application of "g" to "y" has to be done before the resulting function from "f x" is applied.

Abstraction of variables is a pattern directed approach. The input PSASL expression is separated into its subexpressions and the variables are abstracted according to the shown abstraction algorithm.

TABLE 1. Abstraction Algorithm

[x] E1 E2 = S ([x] E1) ([x] E2)	for all E1, E2
[x] E = K E	for any object E = x
[x] x = I	
[x] E x = E	for any expression E
[x] E1 E2 = B E1 ([x] E2)	for any E1, E2 where x occurs in E2
[x] E1 E2 = C ([x] E1) E2	for any E1, E2 where x occurs in E1

SOURCE: Turner, D.A. "Combinator Reduction Machines", Workshop on High Level Computer Architecture, (1984):5.29

For example, consider a simple PSASL expression involving just the variable itself, say x. Then abstracting x from x, written as [x] x, results in I, identity combinator. A more complex expression involving two subexpressions E1 and E2 where E1 contains an occurence of variable x leads to

[x] E1 E2 = C ([x] E1) E2

The resulting combinator code from the abstraction algorithm grows quadratically with every variable abstracted; thus optimisation rules involving S', C', B' combinators are defined by Turner (1984). The definitions of the combinators used by Turner and in this implementation are given in Table 2.

TABLE 2. Definitions of the Combinators

S f g x = (f x) (g x)	S' k f g x = k (f x) (g x)
K f x = f	B' k f g x = k (f (g x))
I x = x	C' k f g x = k (f x) g
B f g x = f (g x)	
C f g x = (f x) g	
Y f = f (Y f)	

SOURCE: Turner, D.A. "Combinator Reduction Machines", Workshop on High Level Computer Architecture, (1984):5.29

The combinator code using the optimisation rules grows at worst linearly in size of the computed code. Thus, the example of "fac4" with optimisation rules becomes

C I 4(Y (B (S (C' cond (C eq 0)1))
(B (S times)(C B C minus 1)))))

The complete compilation of PSASL is achieved in three steps:

(1) Formal parameters are removed via bracket abstraction. Variables are abstracted from function definitions by an application of the abstraction algorithm.

(2) Recursion is removed. The Y combinator is introduced into an expression of the form " ... where f = E " when E contains f. The converted form is " ... where f = Y ([f] E)." (Turner,1984:5.28)

(3) All where constructs are eliminated. An expression "E_1 where x = E_2" is replaced by "([x] E_1) E_2".

The following example illustrates the compilation algorithm.

The input PSASL expression:

suc 3 where suc n = plus n 1

Abstracting the variable "n" from the inner expression:

[n] plus n 1 {E1 = plus n E2 = 1}

C ([n] plus n) 1

C plus 1

There is no recursion, so next step is removal of the where clause:

(C ([suc] suc) 3) C plus 1

([suc] suc 3) C plus 1   {E1 = suc 3   E2 = C plus 1}

(C I 3) C plus 1

Data Structures

The result of the compilation algorithm is a structure containing only combinators and constants. Note that function names are constants. This code is used to build a left leaning binary tree as the underlying representation for the PSASL expression. An interior node with left subtree f and right subtree x is used to represent "f x." The importance of such a structure is that at any time the function to reduce is indicated by the leftmost leaf in the tree. For example the combinator tree for the example code is shown in Figure 1.

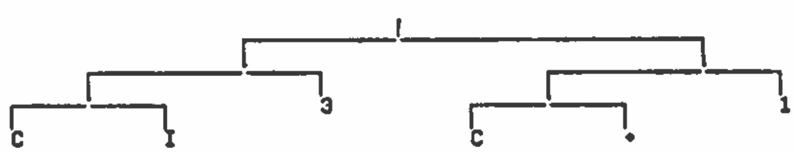


FIGURE 1. The combinator tree of (C I 3) C plus 1 for the PSASL "suc" expression.

The tree structure is implemented using the 'flavors' facility of the ZetaLisp. Operations on the structure are defined via 'defmethods' in flavors. Programs that use flavors are cleaner and conceptually easier to follow;

however, they may not be as efficient as the code written to take advantage of Lisp machine architecture such as cdr coding. The sacrifice of somewhat perhaps a lesser efficient code was made for the clarity in program understanding and maintenance.

Reductions

This thesis implements two kinds of reductions: graph reductions with sharing and tree reductions with copying. Both reductions use normal order evaluation and both start with the same combinator tree. As the reductions progress, one reduction shares the common subexpressions while the other copies them. Both types of reductions use the same data structure with slight variations due to the different display algorithms. Although the reductions are based on Turner's combinator reduction machine, the mechanism that determines the sequence of reductions is slightly different. The difference is that Turner's improved reduction machine involves two pointers for the sequencing of reductions. This implementation uses the previous version of the reduction machine where the order of reduction is given via a left ancestor stack.

The PSASL expression, which is in form of a binary tree after compilation, is reduced according to the combinator or function that appears in its leftmost leaf. The structure is transformed into a new structure according to the definition of the combinator or function. Consider a simple combinator structure such as

C f g x = (f x) g

The C combinator applied to functions "f" and "g" and variable "x" results in application of "f" to "x" and, application of the resulting function, i.e. "f x" to "g". The corresponding graph transformation is shown in Figure 2. Such reductions are continued until the whole structure simplifies to a structure in which no further applications are possible. Typically this will result in a single node which holds the value for the input expression.

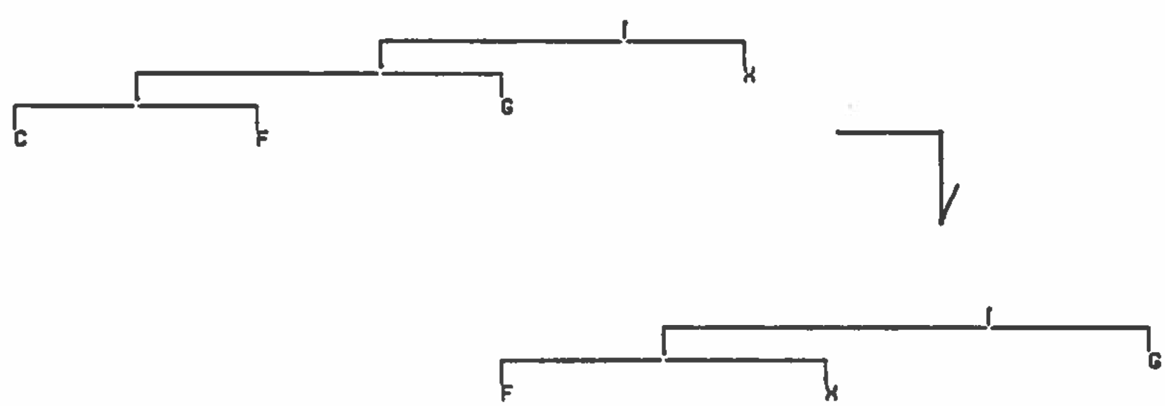
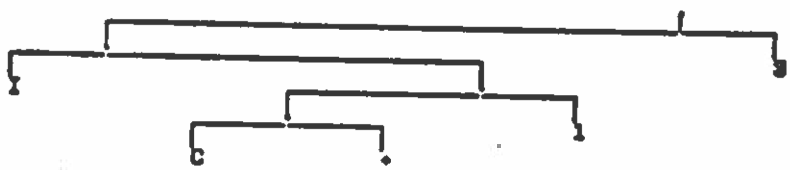


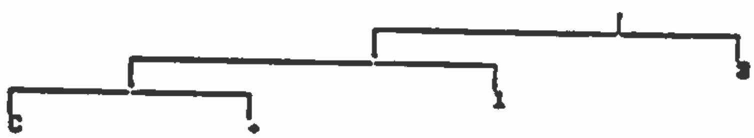
FIGURE 2. Reduction on the C combinator,
i.e. C f g x = f x g.

The reductions for the "suc 3 where suc n = n + 1" are shown below. The initial structure is given by the combinator expression derived earlier.

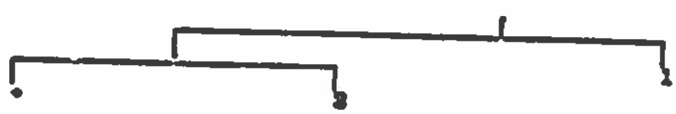
C I 3 (C plus 1)



I(C plus 1) 3



C plus 1 3



plus 1 3

Reductions with Copying

The most important characteristic for measuring efficiency of these reductions is the necessity to copy parts of a structure. For example, given the expression S f g x, the combinator tree will be tranformed to the tree for f x (g x) where the tree that represents x will be copied (See figure 3).

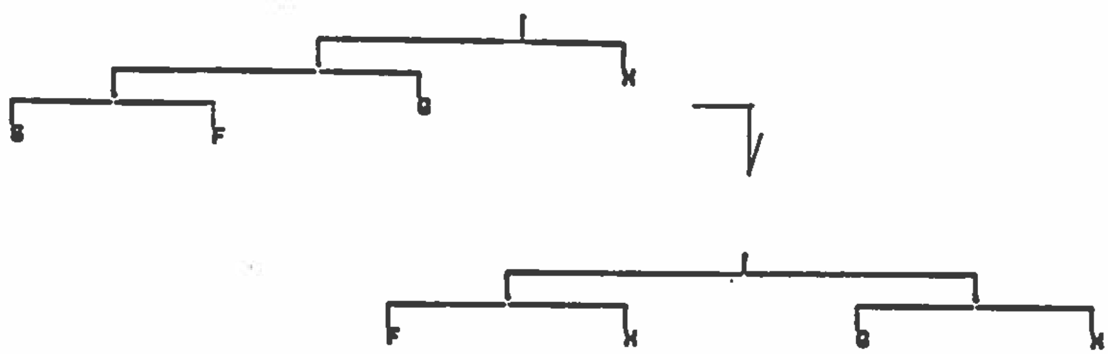


FIGURE 3. The S combinator in copied reductions, i.e. S f g x = f x (g x).

Other combinator reductions that require a copy operation are S' and Y. (See Figures 4 and 5).

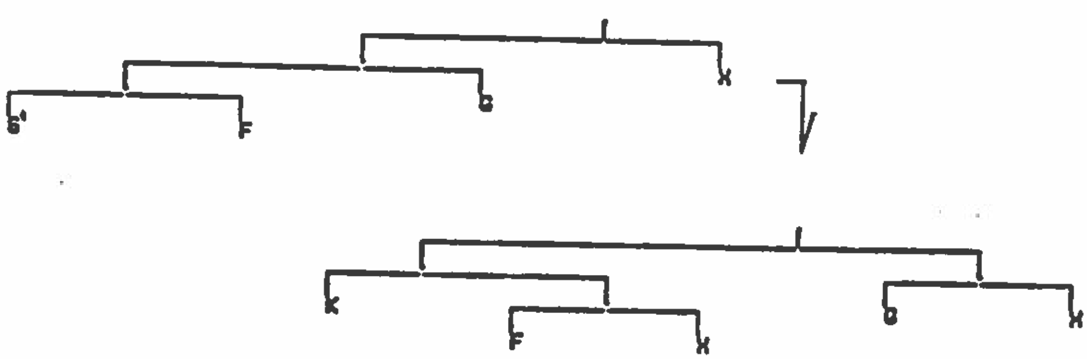


FIGURE 4. The S' combinator in copied reductions, i.e. S' k f g x = k f x (g x).

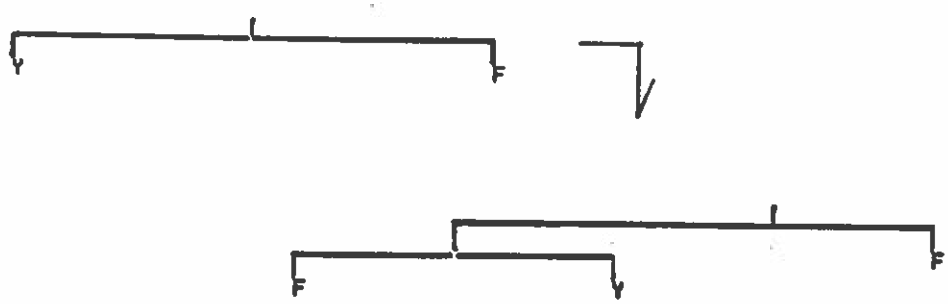


FIGURE 5. The Y combinator in copied reductions,
i.e. Y f = f Y f.

Reductions with Sharing

In these graph reductions the need for the copy operation is eliminated for Y, S, S' combinators. This is achieved through sharing of the common argument in the transformed structure rather than copying it. The result is a mechanism with less use of memory and most important, more efficient reductions.

In graph reductions, sharing of a subtree is achieved with S and S' reductions by a simple pointer manipulation. For example, in describing the reduction S f g x = f x (g x), the second x is not duplicated. Instead a node is created for the second x which points to the original x. Therefore, there is only the original x and it is shared by both f and g. (See figures 6 and 7).

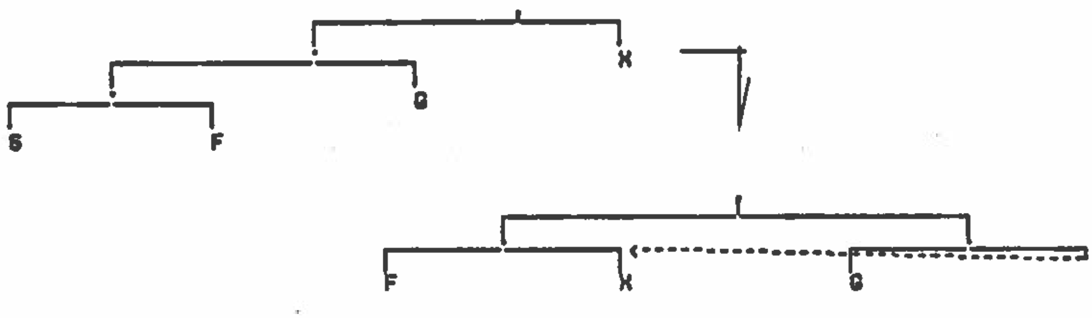


FIGURE 6. Shared reductions with the S combinator, i.e. S f g x = f x (g x)

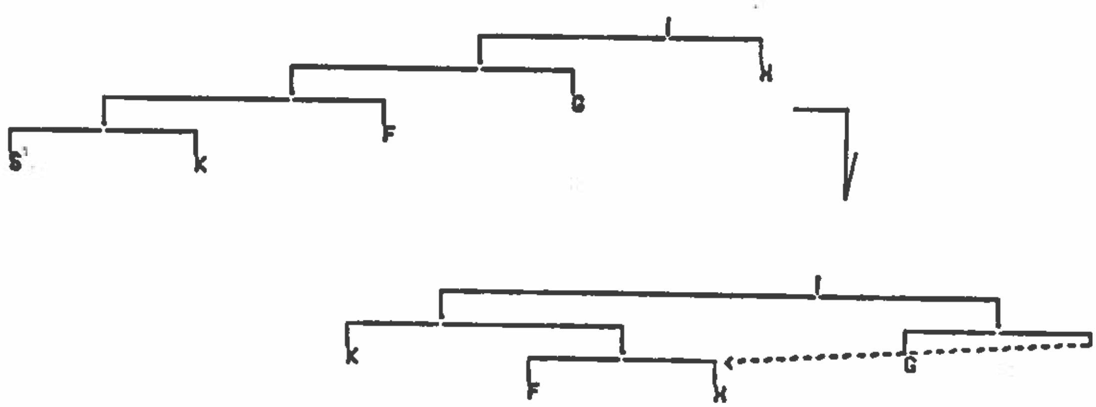
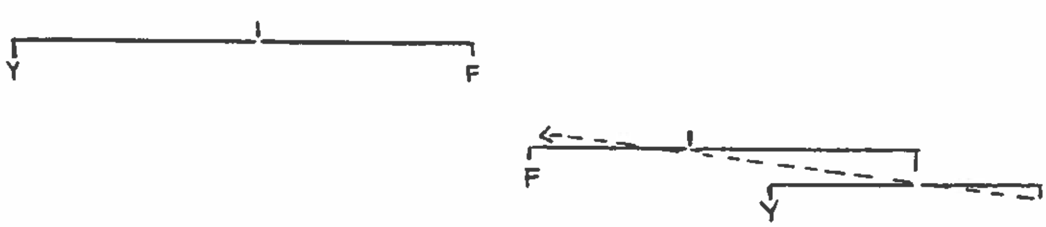


FIGURE 7. Shared reductions with the S' combinator, i.e. S` k f g x = k f x (g x)

Sharing of a subtree for recursive expressions involving the Y combinator is less intuitive. Given the definition of Y as: Y f = f Y f, the expected visual correspondance would be:



However, the reductions of Y is diagramatically defined as shown in figure 8 where the f on the right hand side has a pointer to itself. This corresponds to the definition Y f = f rather than Y f = f Y f.

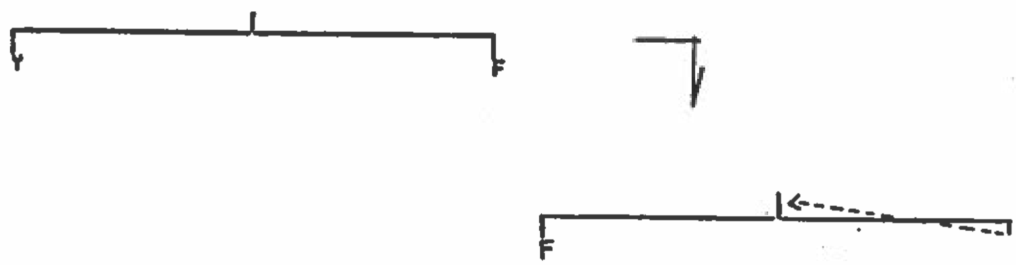


FIGURE 8. Shared reductions with the Y combinator, i.e. Y f = f Y f.

Reductions with sharing work because of referential transparency. Once appropriate links are established to the common structures, the functions at the root of the links share the effects of evaluation. Thus, if e is a common subexpression in "plus (f e) (g e)", e is evaluated once (when required by f), and then g automatically gets the reduced version of e, since it points to the same node f points to. This results in shared structures being reduced only once (a property of applicative order evaluation). Also shared structures are reduced only when necessary (a property of normal order evaluation). The extensive example given in the next section demonstrates this property. Thus graph reduction emerges as a very powerful technique where the efficiency of applicative order is provided together with a demand-driven processing evaluation for arguments of a function.

Graphical Display of Reductions

Two algorithms are implemented to draw trees and graphs on the bit mapped display of the Symbolics 3600.

Displaying Copied Reductions

The algorithm to display the binary tree uses two passes over the structure. In the first pass information on number of descendants for each node is obtained. In the second pass, the nodes of the tree are displaced horizontally from the parent node according to the number of descendants on the left and right subtrees. This provides flexible positioning of nodes on the screen, which is divided N parts, where N is the number of descendants. This enables a complete display of a tree with any number of nodes; however, there is the danger of having all the information scrunched at the lowest level if the tree is large (experince shows the display starts to look bad at 80 nodes). This limitation is due to the physical screen size of the Lisp machines--which is actually quite spacious at 1088 by 748 pixels-- and is not a limitation posed by the algorithm.

Displaying Shared Reductions

The algorithm to display graphs is also a two pass algorithm. In the first pass, similar to the tree display algorithm, information on the number of descendants is obtained; however, more information is required to display a graph. The extra information, of whether a node is a shared node or a root of a shared subtree of subgraph, is determined in pass one.

In the second pass, the algorithm displays the graph according to the number and type of its descendants. The backpointers and self pointers (pointers to shared structure) are drawn as a dashed line to distinguish them from the other links of the structure.

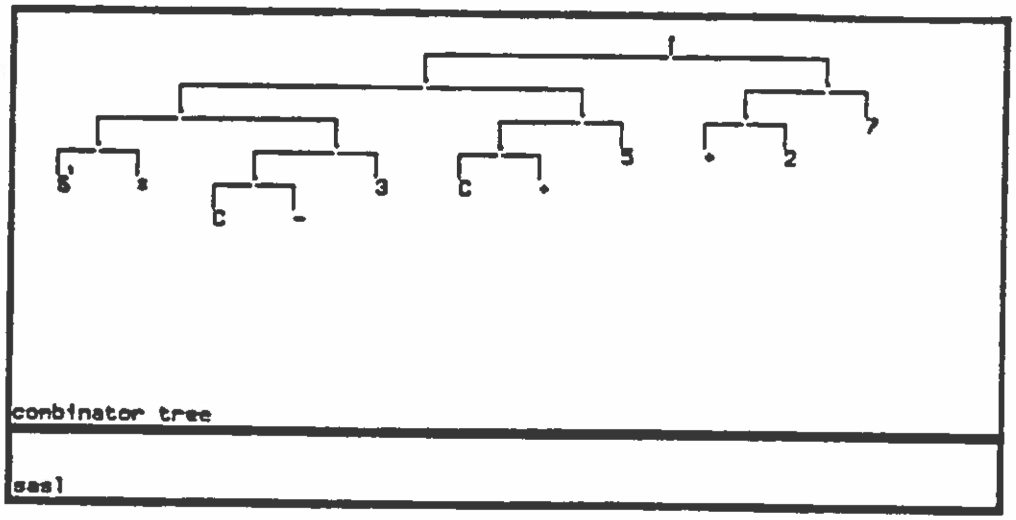
An Example of Copied Reductions

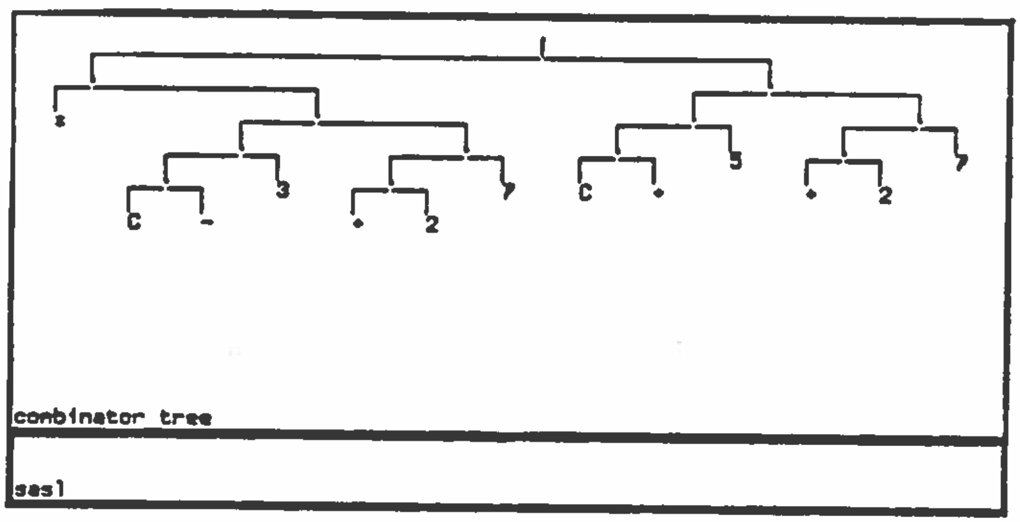
Consider the PSASL expression:

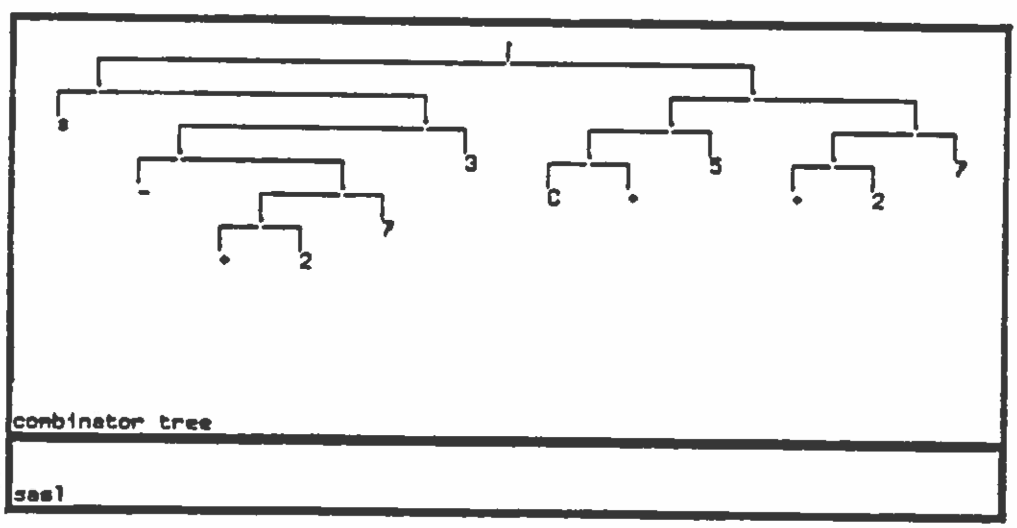
times (minus x 3) (plus x 5) where x = (plus 2 7)

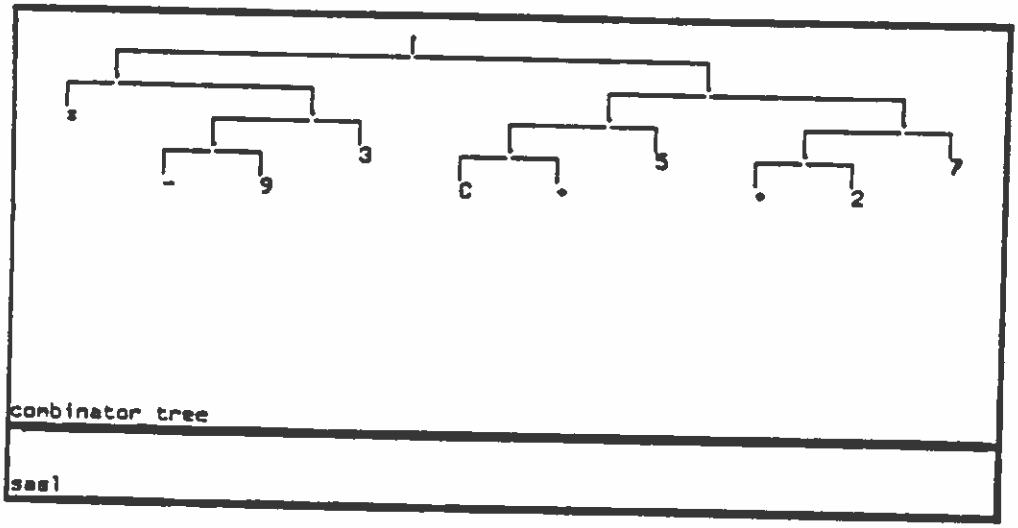
The diagrams given below start with the combinator tree representation of the PSASL expression and continue with transformations that are taken at each step of the reduction until the result is reached. Note that the subtree representing the value of x, i.e. plus 2 7, is copied and evaluated more than once. The duplication of "plus 2 7" can be seen in second diagram while its evaluation is seen in diagrams 4 and 7.

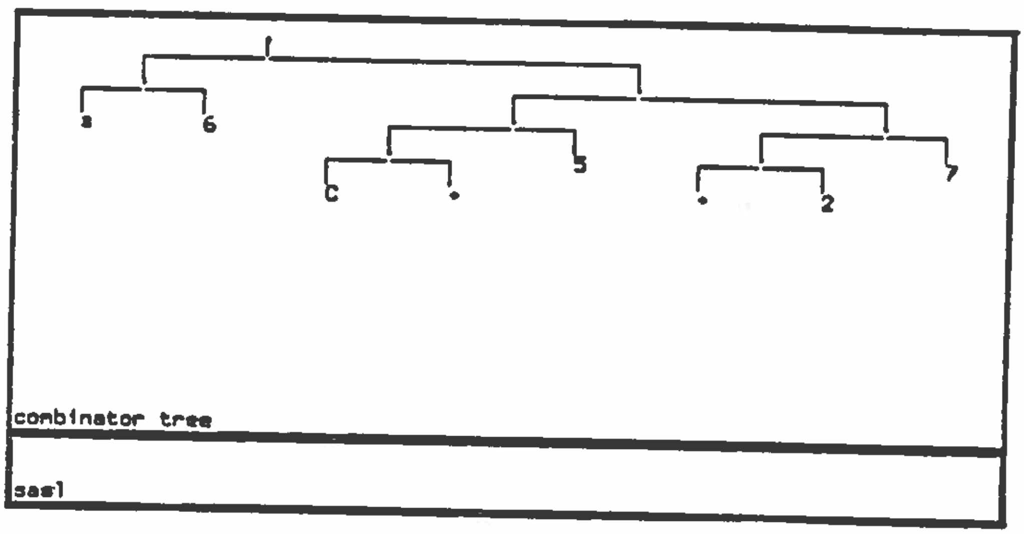
diagrams 4 and 7.

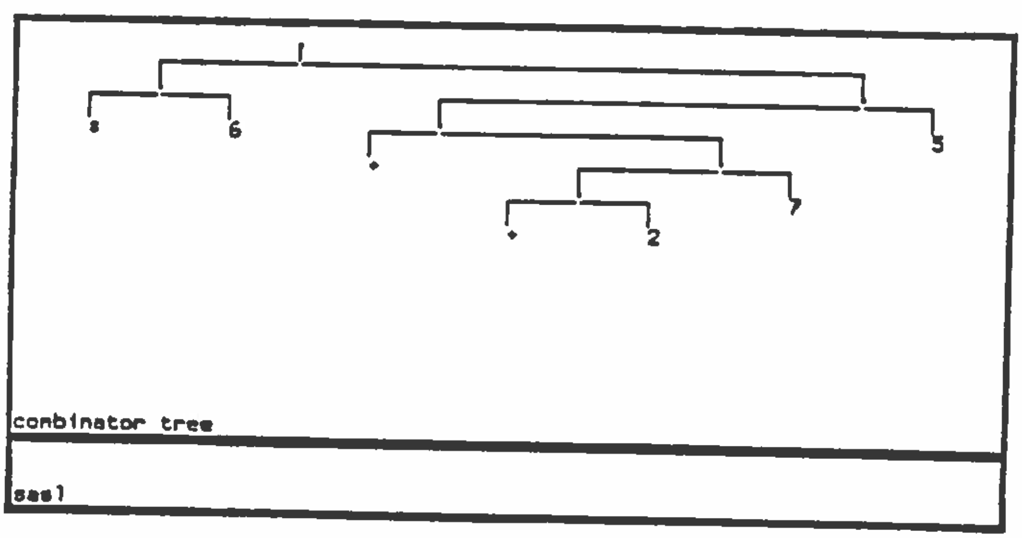


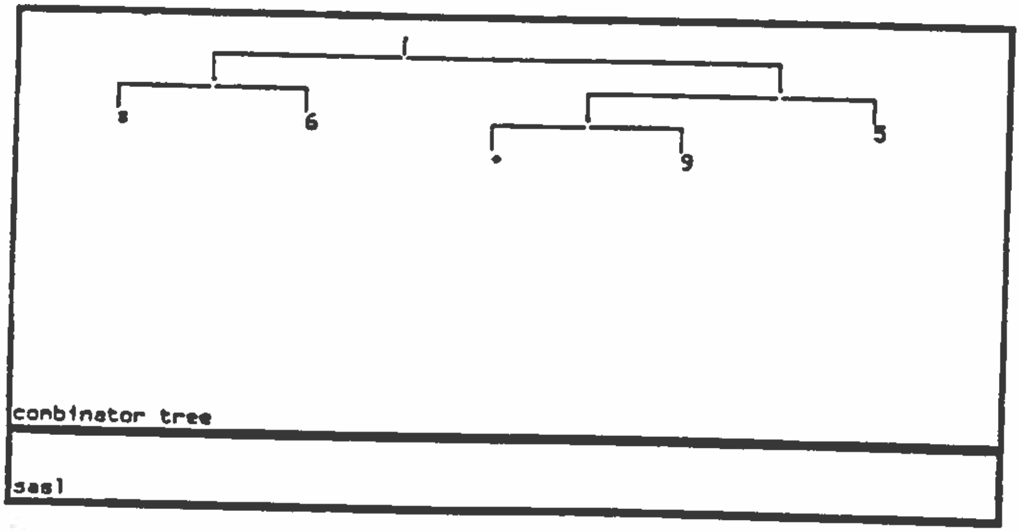


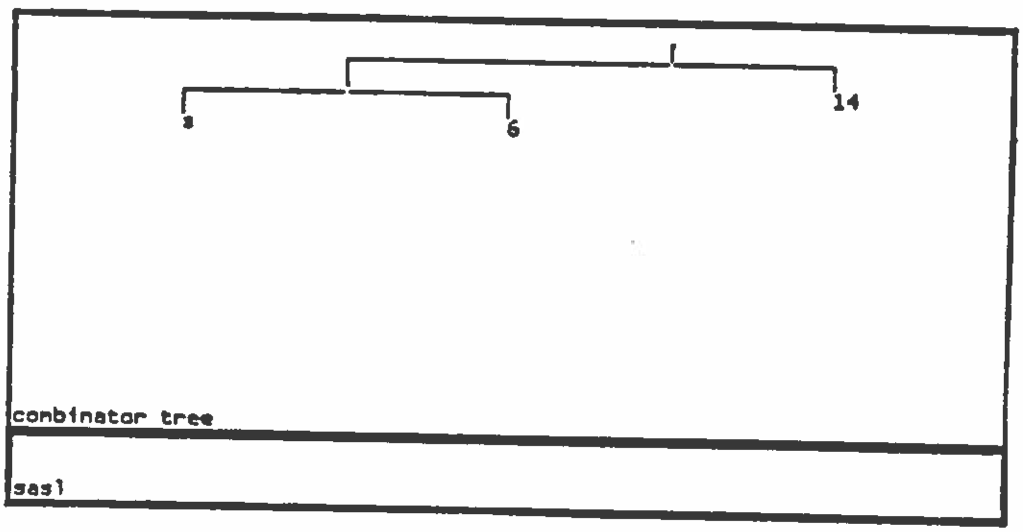


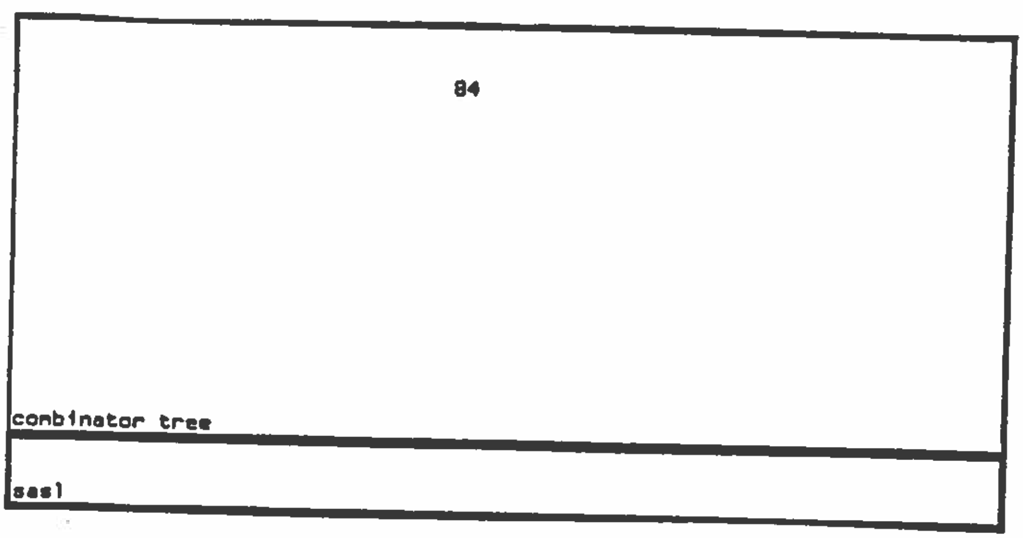

combinator tree
sasl


combinator tree
sasl


combinator tree
sasl

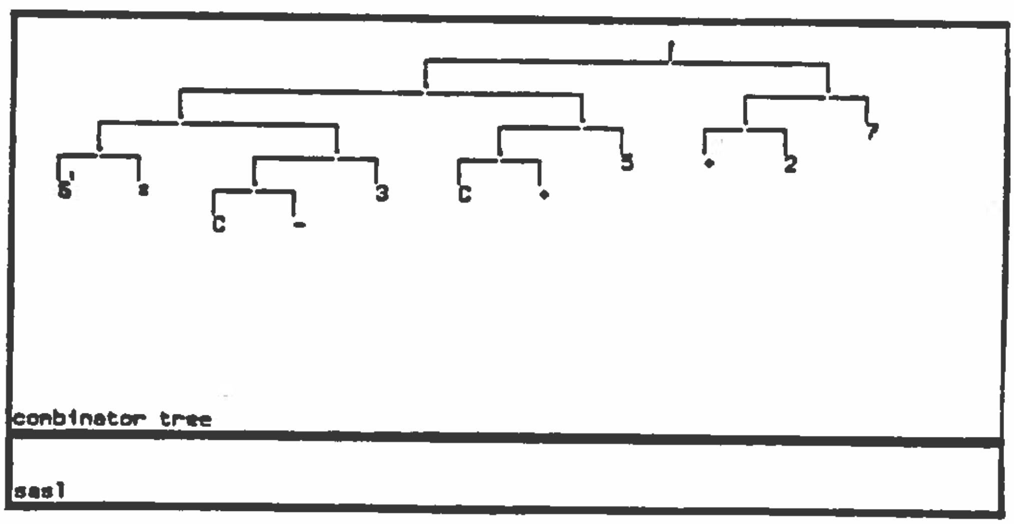

6
9
5
combinator tree
sasl

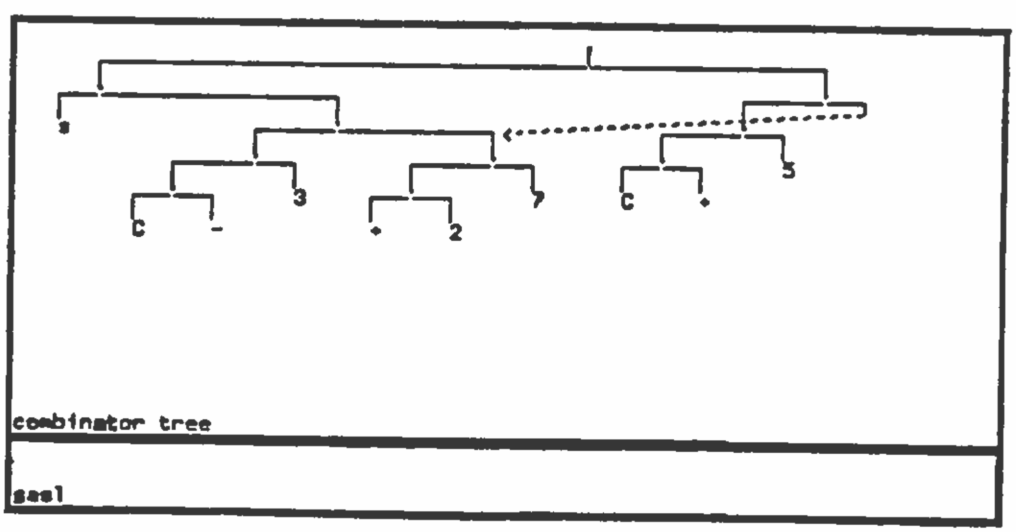

14
6
combinator tree
sasl

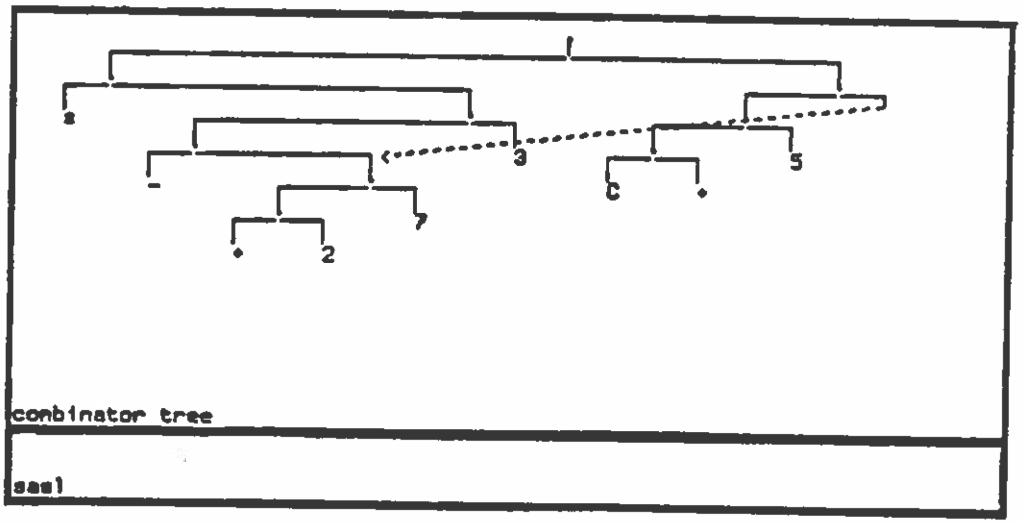

84
combinator tree
sasl

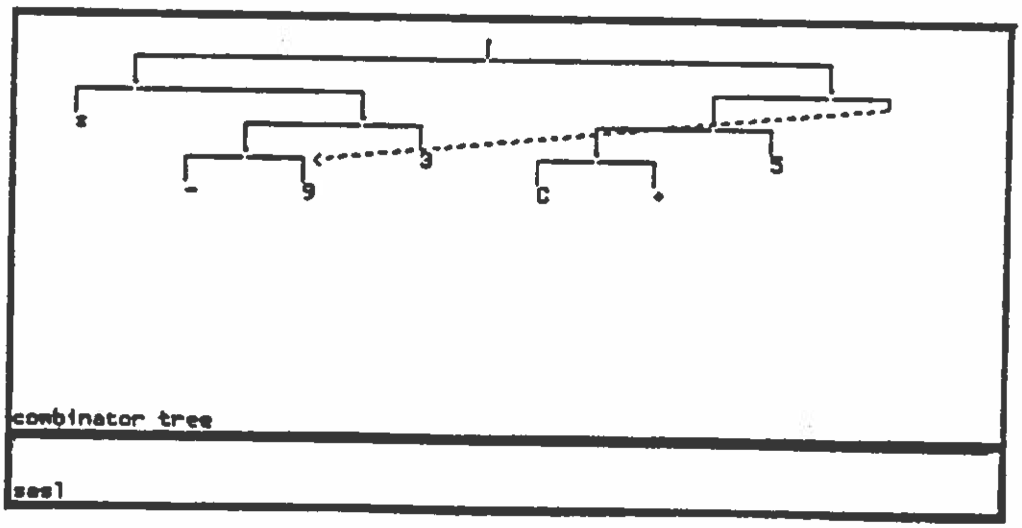
An Example of Shared Reductions

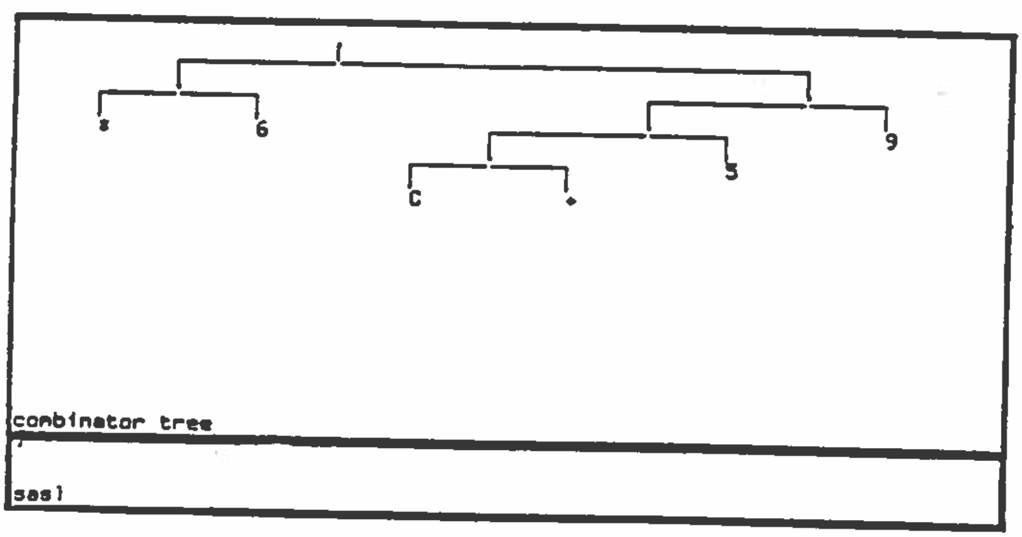
The diagrams given below show each state of the reduction of the same expression in which common subexpressions are shared instead of copied.

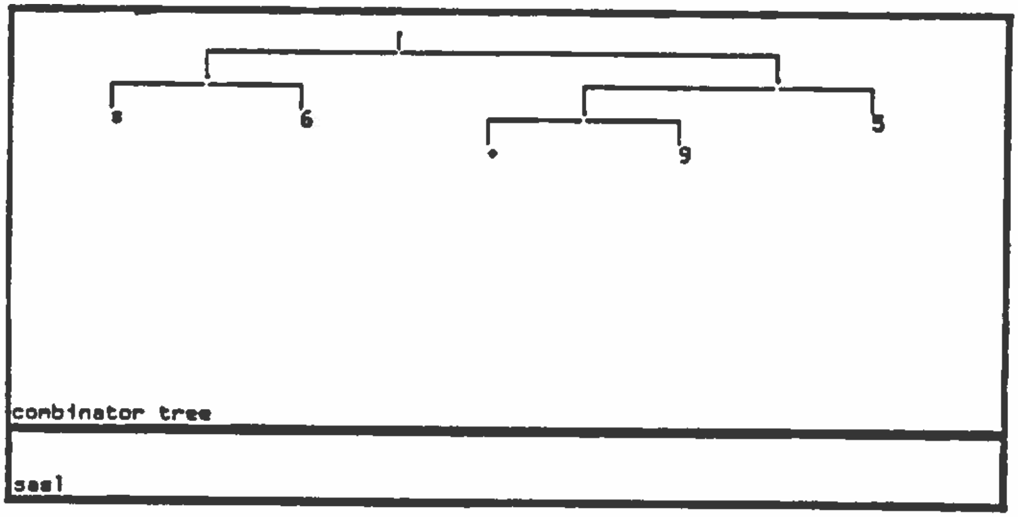


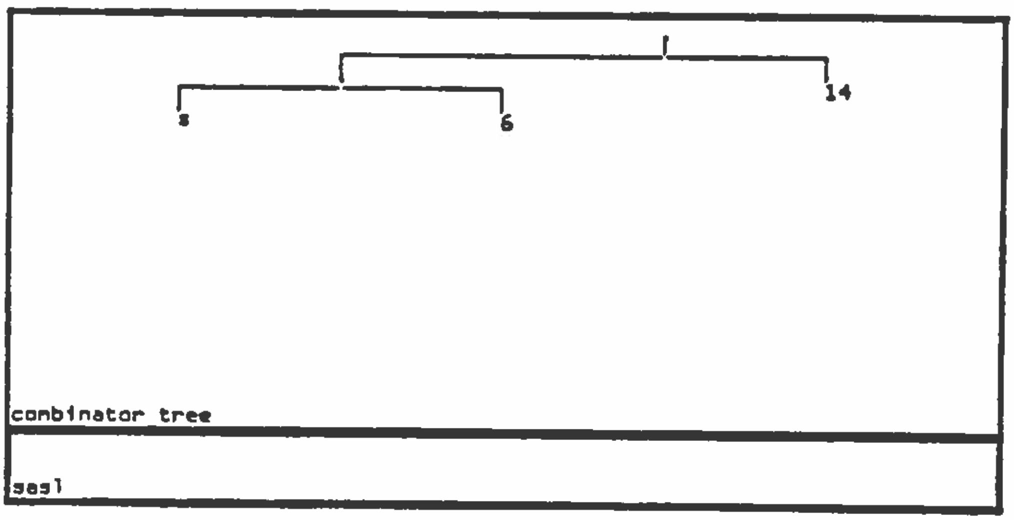


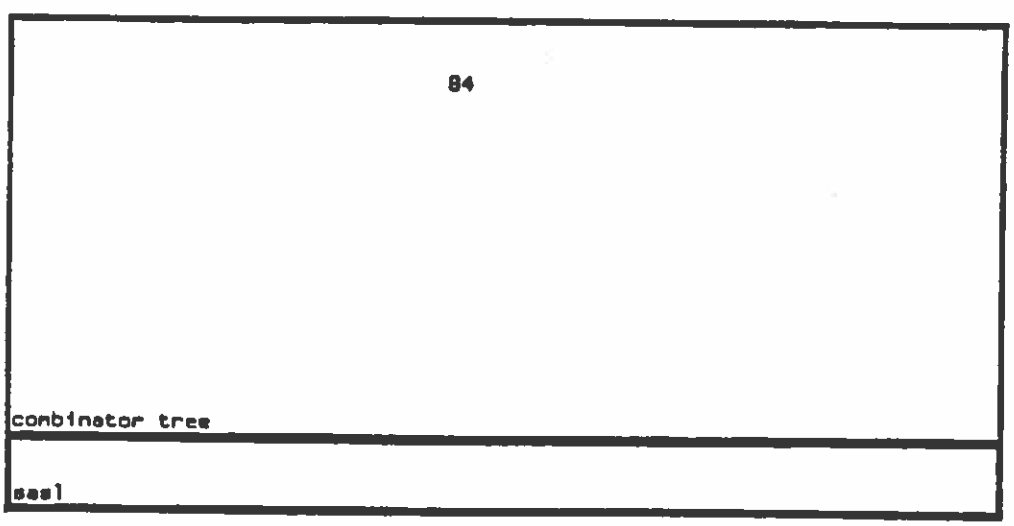

combinator tree
sasl


combinator tree
sasl


combinator tree
sasl


6
9
5
combinator tree
sasl


14
6
combinator tree
sasl


84
combinator tree
sasl

CHAPTER IV

EXPERIMENTS

A number of functions with varying computational expense are used as test cases. These test cases were executed by both copied and shared reduction engines. Measurements obtained are contrasted with the expectation of learning which implementation is more efficient and how optimisations affect the number of reductions.

Test Cases

The following PSASL programs where used as benchmarks:

Factorial: Computes the factorial of 15 using a recursive method.

```
fac 15 where fac n = cond (eq 0 n) 1
                     (times n (fac (minus n 1)))
```

Fibonacci: Uses the recursive definition to compute the 10th fibonacci number.

```
fib 10 where
        fib n = cond (eq 0 n) 1
                (cond (eq 1 n) 1 (plus (fib (minus n 1))
                                       (fib (minus n 2)))
```

Permutations: Counts the 7 element sequences chosen from 11 without repetition via the difference equation P(n,r) = n *

P(n-1,r-1).

```
perm 11 7 where
           perm n r = cond (eq 0 r) 1
                      (times n (perm (minus n 1)
                                     (minus r 1)))
```

Combinations: Computes the binomial coefficient C(4,2) using the PASCAL Triangle relation C(n,r) = C(n-1,r-1) + C(n-1,r).

```
comb 4 2 where
         comb n r = cond (eq 0 r) 1
                    (cond (eq 0 n) 1
                      (plus (comb (minus n 1) (minus r 1))
                            (comb (minus n 1) r)))
```

Twice: Computes the curried function twice defined by Turner. (1984:5.27)

```
(twice twice twice suc 2
                where twice f x = f (f x))
                        where suc n = plus n 1
```

Note that "twice f x " is f(f(x)); the answer here is 64.

Power: finds the 7 th power of 3.

```
pow 3 7 where
         pow n m = cond (eq m 0) 1
                   (times n (pow (minus m 1)))
```

Binomial Coefficient: Computes C(10,3) coefficient in the binomial expression.

```
(bin 10 3  where
   bin n r = div (fac n) (times (fac (minus n r)) (fac r)))
      where
         fac z = cond (eq 0 z) 1 (times z (fac (minus z 1)))
```

Here we use an algorithm different from "combinations" for the same problem.

Prime: If a number is prime returns the number otherwise returns the first divisor. Determination of primeness is done by test division starting from number 2.

```
prim 61 2
  where prim n f = cond (eq 0 (mod n f)) f
                        (cond (eq n f) n
                        (prim n (plus f 1))
```

Ackerman: Computes a version of Ackerman's function with arguments 2 and 3 (Turner,1982).

```
A 2 3 where
       A x y = cond (eq 0 x) (plus y 1)
                (cond (eq y 0) (A (minus x 1) 1)
                      (A (minus x 1) (A x  (minus y 1)))
```

Arvind: A test to show the difference between shared and copied reductions, through evaluation of arguments once or more than once. (Arvind, 1984:5.2)

```
(plus (g 3) (g 4)
  where
   g y = f (times 2 2) y)
    where
     f x y = (plus (times x x) (times x y)))
```

Measurements

The measurements taken include the following:

(1) Number of applications of each primitive, function and combinator during a reduction,

(2) Number of nodes used for evaluation of memory requirements,

(3) Number of nodes copied (only for non-shared reductions),

(4) Time taken for a reduction.

Copied and shared reductions were executed for each of the given test cases. This enables comparisons of efficiency of one method over the other in terms of number of reductions and storage requirements. Further, the graph reductions with and without optimisation rules (combinators S', C', B') are compared in terms of reductions and storage requirements. All measurements are taken with the display facility turned off. The time measured is only the time it takes to execute the reductions and does not involve abstracting the variables and building the internal representation for the PSASL expression.

Results

Tables 3 and 4 show the data obtained for the test cases from copied and shared reductions respectively. Results obtained vary greatly depending on the test cases. One conclusion that can be made is that deeply nested recursions requiring a lot more computation produce more drastic figures in terms of number of reductions done and nodes used.

TABLE 3. Measurements from Copied Combinator Reductions Optimisation Rules S', C', B' are used

function	S	K	I	B	C	Y	S'	B'	C'	+	-	*	/	total reductions	used nodes	copied nodes
factorial	31	-	1	46	257	16	-	-	16	-	225	15	-	639	1915	1877
fibonacci	320	-	1	496	2310	177	178	-	320	88	1813	-	-	6341	29175	29111
permutations	4	-	1	20	14	4	3	-	10	-	9	3	-	76	297	241
combinations	27	-	1	45	68	15	4	-	16	-	61	13	1	281	946	878
twice	24	-	56	22	19	-	-	-	1	16	-	-	-	138	284	254
power	8	-	1	37	45	8	7	-	15	-	28	7	-	172	527	479
binomial	43	-	2	68	203	23	4	-	25	-	173	21	1	609	1904	1832
primes	119	1	2	298	3601	60	238	-	178	3540	-	-	-	8334	34985	34923
ackerman	296	-	1	777	793	187	513	-	428	78	363	-	-	4028	47664	47570
arvind	2	-	6	5	4	-	3	-	-	3	-	10	-	33	87	45

TABLE 4. Measurements from Shared Combinator Reductions Optimisation Rules S', C', B' are used

function	S	K	I	B	C	Y	S'	B'	C'	+	-	*	/	total reductions	used nodes
factorial	31	-	1	17	33	1	-	-	16	-	15	15	-	177	165
fibonacci	320	-	1	178	499	1	89	-	320	88	176	-	-	2632	2110
permutations	4	-	1	15	8	1	3	-	8	-	5	3	-	60	97
combinations	27	-	1	20	15	1	4	-	16	-	13	13	1	156	181
twice	9	-	10	19	19	-	-	-	1	16	-	-	-	74	69
power	8	-	1	24	18	1	7	-	15	-	7	7	-	112	133
binomial	43	-	2	27	46	1	4	-	25	-	21	21	1	260	266
primes	119	-	2	181	62	1	238	-	178	59	-	-	-	1257	1432
ackerman	42	-	1	77	59	1	55	-	57	12	26	-	-	456	516
arvind	2	-	6	5	4	-	3	-	-	3	-	5	-	28	59

Table 5 compares copied and shared reductions based on time and memory usage. In general, optimised copied reductions can require four to fifteen times more memory than optimised shared reductions. In terms of number of reductions, optimised copied reductions can result in up to three times more reductions than their optimised graph counterparts.

TABLE 5. Comparison of Optimised Copied and Shared Reductions with Regard to Time and Memory Requirements

function	total nodes		total reductions	
	copied	shared	copied	shared
factorial	1915	165	639	177
fibonacci	29175	2110	634	2632
permutations	297	97	76	60
combinations	946	181	281	156
twice	284	69	138	74
power	527	133	172	112
binomial	1904	266	609	260
primes	34985	1432	8334	1257
ackerman	47664	516	4028	456
arvind	45	59	87	28

Table 6 gives measurements on combinators used without the optimisation rules. In comparing graph reductions of table 6 with the optimised graph reductions in table 4, it can be seen that for small functions such as arvind, twice and permuations, there is no significant improvement. However, computationally more demanding functions show 20%

improvement with optimised graph reductions over non-optimised ones with regards to number of reductions done. The memory requirements for unoptimised graph reductions, on the other hand, is about 20% less than optimised reductions.

Table 7 provides the comparison on time and space efficiency of graph reductions with and without optimisation rules. As mentioned previously optimisation rules result in less number of reductions, thus less time; but more memory usage.

TABLE 7. Comparison of Time and Space Efficiency Between Shared Reductions With and Without Optimisation Rules

function	total nodes		total reductions	
factorial	165	151	193	177
fibonacci	2110	1711	3042	2632
permutations	97	96	72	60
combinations	181	177	178	156
twice	69	70	75	74
power	133	119	135	112
binomial	266	259	293	260
primes	1432	1036	1793	1257
ackerman	516	432	599	456
arvind	59	60	31	28

First columns of each heading refer to shared reductions with optimisations while the second columns under the headings refer to non-optimised shared reductions.

TABLE 6. Measurements on Shared Reductions Without the Optimisation Rules

function	S	K	I	B	C	Y	+	-	*	/	total reductions	used nodes
factorial	31	-	1	49	33	1	-	15	15	-	193	151
fibonacci	409	-	1	588	819	1	88	176	-	-	3042	1711
permutations	7	-	1	27	16	1	-	5	3	-	72	96
combinations	31	-	1	42	31	1	-	13	13	1	178	177
twice	9	-	10	20	20	-	16	-	-	-	75	70
power	15	-	1	47	33	1	-	7	7	-	135	119
binomial	47	-	2	60	71	1	-	21	21	1	293	259
primes	357	-	2	717	240	1	59	-	-	-	1793	1036
ackerman	97	-	1	220	116	1	12	26	-	-	599	432
arvind	5	-	6	8	4	-	3	-	5	-	31	60

CHAPTER V

CONCLUSIONS

Graph reductions are more efficient and faster than copied reductions, which was known prior to undertaking of the thesis. The test cases are based on fairly small expressions although computationally they are comparable to more typical larger expressions. The following conclusions can be deduced at the danger of making oversweeping generalizations:

(1) Optimised graph reductions seem four to fifteen times better in terms of memory needed and two to three times better with respect to number of reductions executed than optimised string reductions. The problems which are most effected seem to be PSASL expressions that contain one or more recursive calls.

(2) Optimised graph reductions in comparison to their non-optimised counterparts, result in about 20% less reductions but about equivalent amount of increased memory usage. This is again seen in computationally expensive expressions. However, for almost all cases there is a general increase in memory usage with optimised reductions. The implication is that although the optimisation rules result in more compact compiled code, at run-time the code

expands and results in higher memory usage than the memory requirements seen in non-optimised reductions. Using optimisation rules is better since in general memory is cheaper than time.

(3) Display of information on a screen is a non-trivial task requiring many considerations such as the appropriateness of the pictorial representation to the user; what parts of the structure should be shown, all or some etc. In this project, some of these issues can be avoided by assuming that only sophisticated programmers who are familiar with trees and graphs would use PSASL. The facility of displaying combinator reductions is helpful in understanding and debugging PSASL programs.

CHAPTER VI

FUTURE WORK

Extensions to the Thesis

A number of extensions are desirable to the current implementation of PSASL:

(1) Extending the abstraction of expressions

(2) Including lists and strings as objects

(3) Including ZF expressions

(4) Modifying the graphical display of PSASL expressions.

The abstraction algorithm, for completeness, needs to handle "multiple" wheres.

The power of SASL comes from its capability of handling infinite streams which are introduced to the system through "lists". This implementation would benefit considerably from inclusion of lists and list primitives such as head, tail, map, etc. In any implementation, input and output is also crucial. Operations to read and write files are desirable as well as operations to manipulate strings .

ZF-expressions, implementations of the set expressions of Zermelo-Frankel, are useful shorthands in describing a general type of iteration over lists. For example, a list spareparts of spare part records can be defined with functions such as name, partnumber and cost. So if "s" is a record then "name s", "partnumber s" and "cost s" will yield the name, partnumber and cost of s respectively. A list of all spare part names can be obtained through the ZF expression

```
{ name x ;  x <- spareparts}
```

meaning "the set of names of all x where x is a spare part." The first part of this expression "name x" is referred to as the <u>body</u>; the latter part involving "<-" is the <u>generator</u> and provides the source for elements of the body, i.e. names in this case. The power of the ZF expressions come with definitions of <u>guards</u> which filter the elements produced by the generator. For example, the names and costs of spare parts which cost less than $15.00 would correspond to

```
{ {name x, cost x} ;  x <- spareparts ;  cost x < 15.00}
```

Thus ZF-expressions provide ease in describing operations on specific parts of lists.

A program that would be nice to compile in PSASL is Turner's solution to the "Eight Queens" problem in SASL using ZF-expressions.

```
queens 8
 where
   queens 0 = [[]]
   queens n = [ q:b; b<- queens(n-1); q<-1..8; safe q b]
     safe q b = all I [ ecks q b i; i<- 1.. b]
     checks a b i = q = b i | abs(q - b i) = i
```

The graphical display of trees and graphs for this implementation is sufficient; however, if the implementation is extended, then possibly a more flexible display method is needed in order to display large structures. A possible technique is to use a mapping from a large virtual screen to the smaller physical screen using the mouse to scroll the display in all four directions. This would require some work with the limited graphics facility and the awkard windowing system on the Symbolics 3600 machines.

A necessary commodity in compiler or interpreter implementations is good debugging facilities. Programmers in PSASL should not need to know about combinators; thus, a non-combinator feedback is desirable as to where the program has gone wrong. The current facility to display PSASL structures during reductions is viewed as a helpful debugging tool for programmers who are familiar with combinators and their definitions. Two goals for the future are:

(1) Develop a method for displaying internal state in user's (source program) terms, not combinators.

(2) If combinators must be used, define a form of graphic feedback, helpful for programmers who are not conversant with combinators.

Further Areas Of Exploration

The implementation technique of combinator-reduction machines lead into further avenues of investigation. An avenue of interest is exploring the potential of parallelism with such a technique.

Functional languages are amenable to parallel operations. The operation of applying one function to another function in PSASL is represented as a juxtaposition of two expressions e1 and e2; in some cases these two expressions can be evaluated in parallel on a multiprocessor. Furthermore, as no side-effects exist, subexpressions can also be evaluated simultaneously. Burton (1984,159:174) suggests a method of annotating SASL-like programs to control parallelism and reduction order in evaluation of functional programs. The annotation he proposes detects whether a given expression can be reduced, partially reduced or abstracted. In Turner's combinator technique, this is detected through replacing a single combinator rule by three other rules. For example, the S f g x is replaced by

$\underline{S}_L$ f g x

$\underline{S}_E$ f g x

$\underline{S}_P$ f g x

These rules correspond to the detection of reduce, make-abstraction and partially reduce operations. The claim is that make-abstraction and partially reduce operations can be performed in parallel. The notion is that with combinators whenever a subexpression is transferred to another processor to be partially reduced, it will be self-contained. The interested reader is referred to the paper by Burton (1984).

APPENDIX

BNF DESCRIPTION OF PSASL

<program> ::= <expr>

<expr> ::= <expr> where <condexp>

<condexp> ::= cond <opexp> (<condexp>) | (<opexp>)

<opexp> ::= <prefix> <opexp> <opexp> | <comb>

<comb> ::= <comb> <simple> | <simple>

<simple> ::= <constant> | (<epxr>)

<constant> ::= <numeral> |<boolconst> | <id>

<boolconst> ::= T | F

<numeral> ::= { digits 0 through 9 }

<id> ::= { letters a through z }

Operators (in order of increasing binding power):

not and or eq noteq less grt lesseq grteq times div plus minus

Built-in functions include:

mod

BIBLIOGRAPHY

Arvind, K.P. "Sharing of Computation in Functional Language Implementations." Proceedings of the International Workshop on High Level Computer Architecture." 1984.

________. Gostelow, and W.E. Plouffe. "An Asynchronous Programming Language and Computing Machine." Technical Report 114a, Department of Information and Computer Science, University of California, Irvine, December, 1978.

Brown, Marc H., and Robert Sedgewick. "Techniques For Algorithm Animation." IEEE Software. (January 1985):28-39.

Burton, F.Warren. "Annotations to Control Parallelism and Reduction Order in the Distributed Evaluation of Functional Programs." ACM Transactins on Programming Languages and Systems. Vol. 6, no. 2 (April 1984):159-175.

Henderson, P. and J.H. Morris. "A Lazy Evaluator." Conference Recordings Of The Third ACM Symposium on Principles of Programming Languages. (January 1976):95-103.

Hindley, J.R, B. Lercher and J.P. Seldin. Introduction to Combinatory Logic. London Mathematical Society Lecture Note Series 7. Cambridge University Press, 1972.

Hughes, R.J.M. "Super Combinators- A New Implementation Method for Applicative Languages." ACM Symposium on LISP and Functional Programming. (1982):11-20.

Johnsson, Thomas. The G-machine. Chalmers Institute of Technology, 1983.

Jones, Neil D., and Steven S. Muchnick. "A Fixed-Program Machine For Combinator Expression Evaluation." ACM Symposium on LISP and Functional Programming. (1982):11-20.

Jones, Simon L.P. "An Investigation Of The Relative Efficiencies Of Combinators and Lambda Expressions." ACM Symposium on LISP and Functional Programming. (1982):150-158.

Landin, P.J. "The Mechanical Evaluation of Expressions." The Computer Journal. Vol. 6, no. 4 (1964):308-320.

McCarthy, J. "Recursive Functions of Symbolic Experessions and their Computation by Machine, Part 1." Communications of the ACM. Vol. 3, no. 4 (April 1960):184 - 195.

Maclennan, B.J. Principles Of Programming Languages: Design, Evaluation and Implementation. CBS College Publishing, New York, 1983.

Richards, Hamilton Jr. "Programming In SASL" Burroughs Corporation Austin Research Center. 3 December 1983.

________. "The Pragmatics Of SASL For Programming Applications." Burroughs Corporation Austin Research Center. Report No. ARC 82-15, 15 June 1982.

Sedgewick, Robert. Algorithms. Addison-Wesley, (1983):171- 223.

Turner, D.A. "Combinator Reduction Machines." Proceedings of the International Workshop on High Level Computer Architecture. (1984):5.26-5.38.

________. "A New Implementation Technique for Applicative Languages." Software-Practice and Experience. Vol. 9 (1979):31-49.

________. "Another Algorithm For Bracket Abstraction." The Journal of Symbolic Logic. Vol. 44, no. 3 (December 1978):67-71.

________. SASL Language Manual. June 1976 (Revised November 1983).

Wadsworth, C.P. Semantics and Pragmatics of the Lambda-Calculus." University of Oxford, 1971.