Mathematical & Computational Applications, Vol. 7, No. 3, pp. 253-262, 2002

A TEMPLATE BASED GRAPH REDUCTION SYSTEM BASED ON COMBINATORS

Abdullah Çavuşoğlu[1] , H. Haldun Göktaş[1]& Necla Vardal[2]
[1]Gazi University Ankara,Turkey. [2]Baskent University Ankara,Turkey
abdulc@tef.gazi.edu.tr, haldun@tef.gazi.edu.tr, neclavardal@yahoo.com

Abstract-Graph reduction is one of the important evaluation strategy for lazy functional programming. A combinator is a function that contains no free variables. The idea is based on the fact that, all of the variables in a program can be removed by transforming it into a sequence of combinators which would be drawn from a small pre-defined (fixed) set of combinators (SKI), or which would be drawn from an unlimited number of non-pre-defined set of dynamic super-combinators (SC). We are suggesting a template based algorithm which reduces the stored graph structure of the super-combinators. We can define any produced super-combinators by using templates and use them to perform reduction on the graph. The template based algorithm could be extended and any produced super-combinators can be defined with a predefined template list. The test results show that our approach increases the efficiency of super-combinators algorithms.

Keywords-Graph Reduction, Combinators, Abstract Programming

1. INTRODUCTION

To execute lazy functional programs combinator based graph reduction is used. It involves converting the program to a lambda calculus [3] expression and then to a graph data structure. One method for implementing the graph data structure is that translation of the program to combinators [2]. In this method, all variables are abstracted from the program that is represented as a computation graph, with instances of variables replaced by pointers to sub-graphs that compute values. Graphs are evaluated by, repeatedly applying graph transformations until the graph is irreducible. Then, to execute the program the graph data structure is rewritten.

In [1-2] a rigid combinator based reduction is criticized. Hughes [1] suggests a compilation algorithm that generates efficient combinators -called supercombinators- (SC) than Turners SKI combinator [2]. Hughes method depends on identifying maximally free expressions (mfe) and exporting these as function arguments thereby converting expressions into combinator applications. There are a number of possible optimizations using similar techniques. Here, we are suggesting a technique which generally produces better results than the techniques above. Our primary concern was gaining the computational efficiency in terms of reduction steps rather than storage efficiency – although our algorithm needs less storage than the others- or time efficiency for graph reduction system based on untyped lambda calculus. Improvements referred in [1] have been integrated to our SC implementation. The analysis program that we developed for SC reveals that, a portion of these SC's structure look similar. Using that, SCs can be classified and two SCs with similar tree structures can be expressed by one template. If we assume that, the number of templates is 'k', and number of SCs which has been produced is 'n', and at worst case 'k = n', and for other cases 'k < n'.

2. THEORETICAL BACKGROUND

2.1. The lambda Calculus

The lambda calculus is a simple means of describing the properties of computable functions, effectively treating them as rules. Only a few constructs and simple semantics are required. Furthermore, it's expressiveness is so sufficient, so one can express not only all functional languages but all computable functions as well [8]. The pure lambda calculus contains no constants neither numbers, nor mathematical functions, such as '+', '-' and is untyped. It consists only of lambda abstractions (functions), variables and applications of one function to another. All entities must therefore be represented as functions. Later on, a type theory was also developed [7]. Here, our primary interest was the un-typed lambda calculus with constants, since this is the simplest calculus powerful enough to express basic concepts that we will be dealing with (i.e. functional forms, partial applications, and strictness properties). The details for the syntax of the expressions can be found in [4-5].

2.2. The Reduction Mechanism of Turner's SK Algorithm

The first step in compiling a functional program to a combinator graph is to transform it into an expression in the lambda calculus. The consequent step is to transform lambda calculus to SK Combinators which are a small set of simple transformation rules for functions. Each combinator has a runtime action and a definition associated with it. The definition is used to translate programs from the lambda calculus into combinator form and the runtime action is used to perform reductions to evaluate expressions. S, K, and I are sufficient to represent any computable function. The S combinator, can be called lifted application [9], while the K combinator, called the elementary cancellator, makes a constant function and the I combinator, called the elementary identificator, is the identity function are expressed as:

$S\ x\ y\ z \rightarrow x\ z\ (y\ z)$

$K\ x\ y \rightarrow x$

$I\ x \rightarrow x$

There is one more step to compile a functional program to a combinator graph. How does one map the combinators into a data structure for computation? The data structure of our choice is using a list in order to represent the tree structure. Each node of the tree has a left-hand side, and a right-hand side. These nodes would be either an argument or sub-tree. The leaf of the tree may be a combinator or constant, or operator (+, *, &) or identifier. Using the full Turner's set of combinators can be accomplished by first compiling to SKI combinators, then applying the optimization rules given in [8].

2.3. The Reduction Details of the Hughes' Super- Combinator Algorithm

One method for creating large combinators is using SC compilation. S, K and I combinators can be defined by λ- expressions:

$S = \lambda x.\ \lambda y.\lambda z.(x\ z)(y\ z)$

$K = \lambda x.\ \lambda y.x$

$I = \lambda x.x$

The intention is that these λ- expressions can be used as operators in the reduction machine. The first reason is, they have no free variables inside and secondly, their bodies are in applicative forms, which mean that, they consist of variables and constants.

Therefore any λ- expression with these two properties can become a SC [1]. Unlike the Turner's fixed combinators, SCs are formed by the user's program, rather than from a fixed set of combinators. A lambda expression is normally transformed to a special SC format by assigning it a name, associating the bound variables with it, and setting it equal to the body of the lambda expression.

Hughes mentions that in order to perform fully lazy implementation one more step is needed. When producing SCs, if we take out only free variables then at the reduction part fully lazy evaluation does not seem to be possible. For that reason we have to take out sub-expressions of λ-expression which does not depend on bound variable. These sub-expressions are called maximally free expressions. The resulting *mfe* is replaced by the new parameter name that is allocated. When the whole body has been scanned, the compiler can generate code for new SCs and use them to construct the replacement applicative form [1]. SCs can reduce the manipulations of the graph as well as the graph size by providing customized combinator functions. Dramatic improvements in the execution speed are said to be possible [6].

3.THE TEMPLATE BASED ALGORITHM

In Turner's SKI combinator reduction mechanism, a lambda expression abstracted at phase II and a program result consisting of a tree whose nodes are of the type of combinator (i.e. S, K, I), constant or primary operators. While, in Hughes SC mechanism, the lambda expression is abstracted, *mfe*'s are determined and replaced with parameter names and SCs that are produced. Each produced SC structure tree, is stored with bound variable names.

In our approach, instead of storing each SC structure separately, we try to use a template –holding a possible skeleton structure- that could be used for SC definition. In the program, we do not need to store every SC structure each time: let's assume that the number of SCs which has been produced is 'n', using Hughes reduction technique we need to store n tree structures, however, in our algorithm, if the number of tree structures is 'k', at the worst case we get 'k = n', but for most of the cases we get 'k < n'. This means, a considerable number of structures need not to be saved. A coded example(1) is given below:

```
{ f(3,5) :
  f = { @x.@y.g(x,y) :
        g = { @z1.@z2. h(z1,z2) :
           h = @q.@r. q + r }}}
```

We have a function definition consisting of 3 definition-groups. In the notation '{' denotes def-group and f, g, h are function names and x, y, z1, z2, q, r are function arguments. The six SCs produced by Hughes' method are shown in Figure 1. They all have to be stored, because in the evaluation part, to perform reduction on the graph tree, this description will be used to replace input arguments with the combinator's argument variables. As seen from the Figure 1, the tree structures of C1, C3, C5 and C2, C4, C6 are similar. We used them as two templates to represent 6 super-combinators as shown in the Figure 2.

At the second phase, the inner def-group is compiled first. During the compilation, if we find a lambda definition we take lambda expressions and abstract them in the

following manner: when a tree is visited, *mfe*'s are identified first, (i.e. this can be a single identifier that is free or a list of free identifiers with variables or primary operators), if the *mfe*, which we found in that node exist in the previous nodes then the name of previous

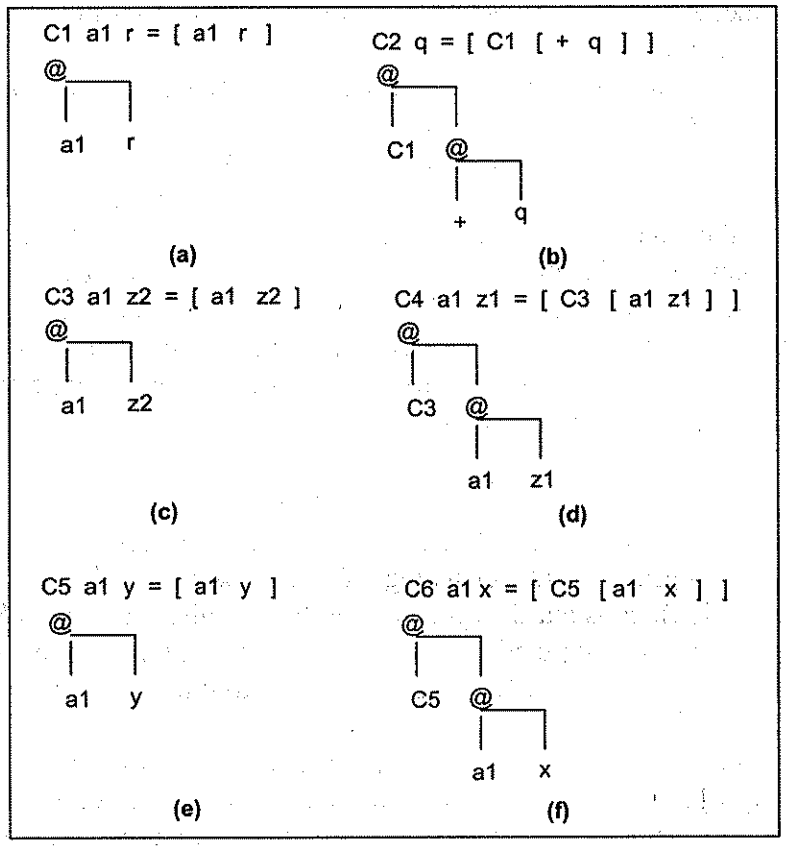

Figure 1. SCs produced by Hughes' algorithm for "example 1"

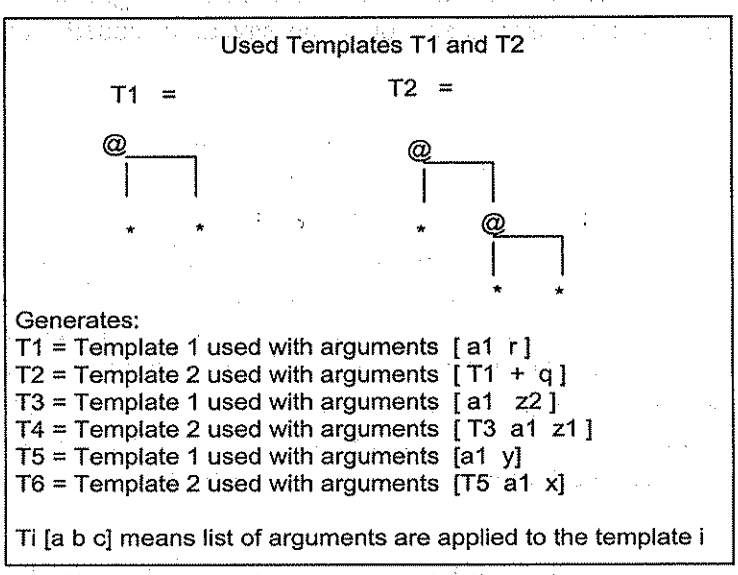

Figure 2. SCs produced using template based method for "example1"

parameters are given. If this is a newly found *mfe*, then it is replaced by a relevant parameter name. After visiting all nodes, the expression's abstracted structure is compared with the template library with the aim to finding a similar structure. If a matching structure is found, then a record that holds a pointer to this structure and, the list of arguments that will be applied to this structure are created. After compiling each definition in the def-group, the def-group's main expression is compiled and replaced with the compiled definitions. The final code consists of records that show which structure is used with which argument.

Evaluation mechanisms look similar in both approaches. Hughes takes arguments and uses SC definitions to replace them in relevant expressions while in our approach the T operator is applied to the taken arguments within the structure for Ti with respect to arg-list. In the arg-list there may be another T operator to be applied to arguments within another structure.

In our approach, we treat templates like machine instructions and we use one T operator to reduce the expression. In our study we used three types of operators. Strict operators require some or all of their arguments to be evaluated before the operator can produce an answer. For example, '+'and '*' are some of the strict operators. The '+' operator first takes 2 arguments (these may be variables which do not need to be reduced or may be a sub-tree containing nodes need to be reduced) and when the sub-graph is eventually reduced, a combinator within the evaluated sub-tree will return a value. The '+' combinator also rewrites the node which was the parent of the node containing the '+' combinator, so that if the sub-tree is shared the evaluation need only be performed once.

Our template based Combinator Reduction System (CRS) can use totally strict operators that perform computations and return results and can use partially strict operators primarily for conditional branching (i.e. 'if'). The IF combinator evaluates it's first argument, then selects the second argument if the first argument is true, or selects the third argument if the second argument is also false. CRS also uses non-strict operator T in order to apply the argument list to the templates and performs reduction that was done by SC definitions used in Hughes method. In CRS we try to generalize the SC definitions by using the T operator and predefined templates instead of producing and using several SC definitions. This reduces the amount of used memory. The CRS algorithm was also improved after analyzing the test results. This time the T operator is not needed in order to perform reduction in the Graph. In CRS implementation the T operator takes index number and applies argument lists to the template, related with that index number. CRS directly does the same process when it finds an index number. So, the resources are not wasted to reduce the T operator.

4.TEST RESULTS

In this section, a number of experimental tests and their corresponding results to reveal the difference between the SKI combinator and the Hughes SC graph reduction algorithms; along with optimized algorithms that were mentioned in [1] are presented. The test results guided us to modify our generator and the interpreter. The concept was to classify the SCs with respect to their tree structure and not to store each definition separately. We used templates to express SCs and to perform their jobs.

As stated above, the templates act like instructions and perform reduction with the help of operator T and argument list. When we find the T operator on the graph, we take the index number of template and argument list. An argument list is applied to the template then the result is rewritten. We obtained test results with our initial program then we have finalized our program by refining and improving the first test results. The test section contains two sub-groups: which are SKI v. SC and OSC v. CRS. For the tests we have developed a minimal expression language. A parser for this language has been implemented and tested. Selected test metrics are the numbers of :

- Combinators that are produced and reduced
- Operators that are reduced
- Replaced nodes
- Called procedures that produces reduction and rewrites
- *Mfe*'s that are produced

Table 1 lists the description of test programs used for comparisons, and the number of definition groups. Tests 3, 4, 8, 13, 14 are recursive programs. Table 2 shows the number of combinators that are produced for the tests listed in Table 1 for the Turner's SKI algorithm. It can be seen that, the produced results are usually greater than the Hughes SC algorithm.

Table 1. Test programs used and their descriptions

Test Program	**Def-group Numbers**	**Description**
Test-1	1	Increment given number
Test-2	2	Perform division
Test-3	1	Factorial 5
Test-4	1	Spin-Tree (reverse given list)
Test-5	1	Produce list whose element is the addition of the two inputs
Test-6	3	Return addition of the two numbers (inputs are given before)
Test-7	3	Return the sum of two numbers (inputs are defined in the def-group)
Test-8	4	n^{th} board solution to the 8 Queens problem
Test-9	1	Check first and last element of the list equality
Test-10	2	Return the first argument of two input
Test-11	1	Take three input return the first one
Test-12	1	Increment the first input by one and multiply it with the second input
Test-13	1	Doubly recursive implementation of the Fibonacci sequence
Test-14	1	Return n^{th} element of list s
Test-15	1	It returns the number of recursion taken in computing the n^{th} Fibonacci number instead of the actual Fibonacci number
Test-16	1	A test for recursive calls

Table 2. The comparison of Turner's SKI and Hughes' SC algorithms

Program	Produced Combinator #		Reduced Combinator #		Reduced Operator #		Reduced Procedure #		Rewrite Procedure #	
	Turners	Hughes	Turners	Hughes	Turners	Hughes	Turners	Hughes	Turners	Hughes
Test-1	5	1	1	1	1	1	2	2	3	3
Test-2	29	6	4	6	1	1	5	7	8	13
Test-3	21	1	38	5	18	18	56	29	95	68
Test-4	27	1	69	13	62	62	131	75	231	169
Test-5	7	2	1	2	1	1	1	3	5	5
Test-6	21	6	0	6	1	1	1	7	2	8
Test-7	28	6	1	6	2	2	3	8	6	11
Test-8	1682	9	299355	7752	30320	29580	329675	46378	398030	112441
Test-9	19	1	7	1	6	6	13	7	25	19
Test-10	18	4	2	4	1	1	3	5	4	6
Test-11	9	3	3	3	0	0	3	3	4	4
Test-12	15	2	2	2	2	2	4	4	7	7
Test-13	35	1	97	9	46	46	143	67	236	160
Test-14	48	2	35	6	11	11	46	18	62	34
Test-15	31	1	121	15	58	58	179	85	304	210
Test-16	171	3	47566	8427	7724	7724	55290	19482	74817	39009

The Figures 3 and 4 show produced and reduced number of combinators and a comparison between the two algorithms. The SKI algorithm contains optimizations, for that reason the number of combinators that are produced are reduced at the compilation stage. The difference in the number of produced combinators between the SKI and SC technique becomes greater for the test program 8. This program contains several deep recursions. As a result the number of produced combinators are decreased more than the ones that does not contain recursion. Because of this huge difference in results, test-8 and test-16 are not included in these Figures. When the SC algorithm is optimized [1] the difference appearing in Figures 3-4 becomes even greater.

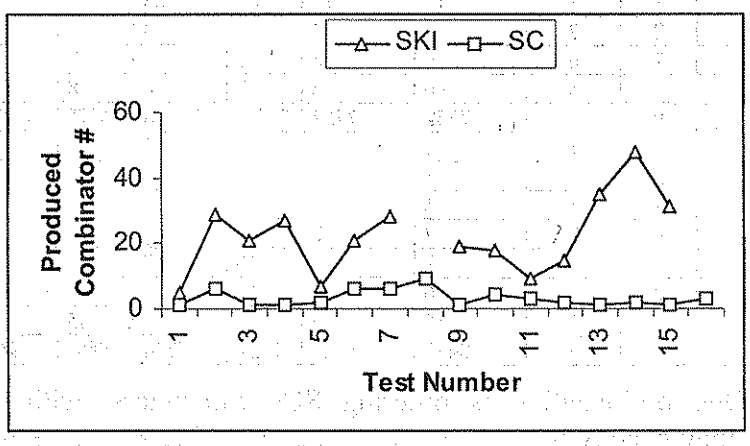

Figure 3. The number of combinators that are produced (SKI v. SC)

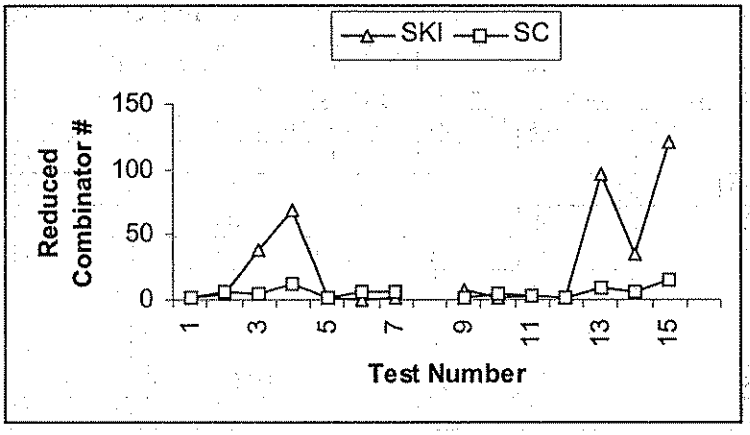

Figure 4. Comparison of the number of reduced combinators (SKI v. SC)

As Figure 4 shows, the number of reduced operators is nearly equal in all test programs. But for the test-8 they are less in numbers when SC is used instead of the SKI. The results suggest that, recursive programs compiled with SC reduction algorithm have lesser number of combinators than the ones compiled with SKI algorithm. Similarly the number of called procedures causing the reductions and rewrite operators are also greater in the SKI algorithm. The optimizations that were suggested by Hughes have been applied to our SC implementation and table 3 shows the test results between initial and the OSC algorithms. These improvements are not new, they are published and applied

Table 3. Comparison between OSC algorithm and the template based CRS

Prog. Name	Produced Combinator #	Template #	Used Template #	Reduced Combinator #	Reduced Template #	Reduced Operator #		Reduced Procedure#		Rewrite Procedure #	
						OSC	CRS	OSC	CRS	OSC	CRS
Test-1	1	1	1	1	1	1	2	2	3	3	5
Test-2	5	4	5	5	5	1	6	6	11	13	13
Test-3	1	1	1	5	5	18	19	29	30	68	70
Test-4	1	1	1	13	13	62	63	75	76	169	171
Test-5	2	2	2	2	2	1	3	3	5	5	9
Test-6	6	2	6	6	6	1	7	7	13	8	20
Test-7	6	3	6	6	6	2	8	8	14	11	23
Test-8	9	9	9	7752	7752	29580	30778	46378	47576	112441	114837
Test-9	1	1	1	1	1	6	7	7	8	19	21
Test-10	3	3	3	3	3	1	4	4	7	6	12
Test-11	1	1	1	1	1	0	1	1	2	3	5
Test-12	2	2	2	2	2	2	4	4	6	7	11
Test-13	1	1	1	9	9	46	47	67	68	160	162
Test-14	2	2	2	6	6	11	15	18	22	34	42
Test-15	1	1	1	15	15	58	59	85	86	210	212
Test-16	3	3	3	8427	8427	7724	7724	19482	19485	39009	39015

to systems. One optimization is ordering SCs parameters, without separating the combinators, if they exactly have the same effect. Detecting such a redundant combinator and eliminating it is necessary because fewer combinators mean fewer reductions to be performed and therefore more speed. Another optimization example is that, the parameters of the combinators can be arranged in any order. If these parameters are ordered with respect to the immediately enclosing lambda expression, it may be possible to obtain less

mfe's. "So, to maximize the size and minimize the number of *mfe*'s of the next enclosing lambda expression, all the *mfe*'s of the lambda expression being compiled which are also free expressions of the next enclosing lambda expression must appear before those which are not " [1].

According to test results -given at Table 3- in Tests 2, 6, 7 the number of templates in CRS are lesser than the number of combinators produced in OSC. But there is an increase at the number of reduced operators, called reduce and rewrite procedures in some test programs. Although we have an advantage on number of produced combinators we have a disadvantage at the number of used procedures for these tests. However, execution speed has been increased for most of the test programs with CRS.

Processed number of combinators for the OSC is equal with the processed number of templates for CRS. Because, although we use one template to express similar structure of SCs, we have to process the same number of templates with different argument lists to reduce the whole graph. There is an increase in the number of reduced operators, called reduce procedures and called rewrite procedures in the CRS program. The Figures 5 and 6 show this increase:

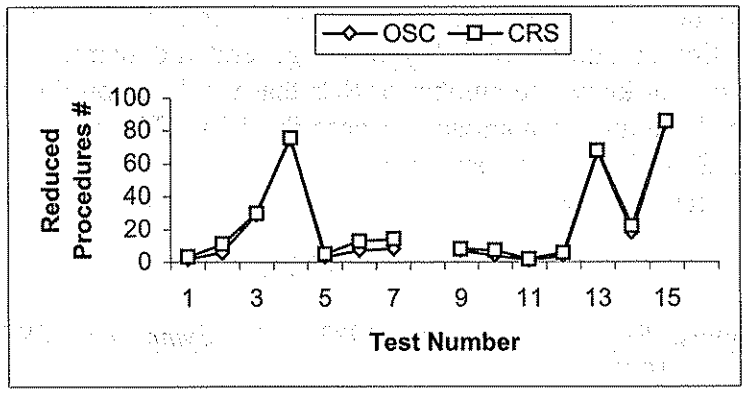

Figure 5. The number of called reduce procedures (OSC v. CRS)

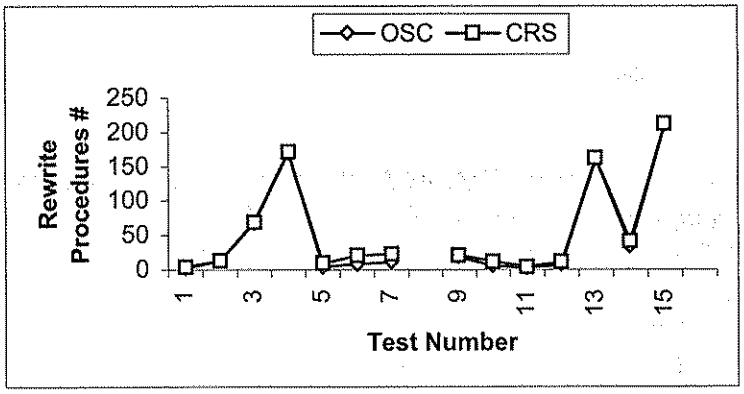

Figure 6. The number of called rewrite procedures (OSC v. CRS)

6. CONCLUSIONS AND SUGGESTIONS

In this study to express SCs shared template structure is used. We classify produced SCs with respect to their tree structure. Then, we use the templates instead of using SC as an operator to perform reduction on the graph. Assume that the numbers of SC that are produced are 'n', and with the template based algorithm the number of used templates is 'k'. At the worst case 'n' is equal to 'k' (produced SC structure are completely different), but for most other cases 'k' is less than 'n' in numbers. By this way, it is possible to define any SCs that are produced with templates. Test results show that, the number of called, reduce procedures and rewrite procedures are higher than the ones obtained from OSC implementation for a small number of test programs. Test results show that, if the produced combinator structure is not similar, template based approach is not useful for these types of combinators.

The most obvious bottleneck of our algorithm is to find matching SC structure in the template list at the compilation part. Implementing the algorithm using templates reduces the stored tree structure for SCs produced. Also, these templates act like an instruction and as a future direction this implementation may be extended to be directly implemented on hardware. Hardware coded templates can be used in the program and the SCs that are produced earlier can be incorporated at compilation stage to speed up the reduction. An unlimited number of SC types are generated dynamically by our algorithm. That is why we do not know the number of SCs that would be produced by our program. But we can use the template approach to describe SCs. CRS reduces the stored graph structure of the SCs. It may be suggested that we can define any produced SCs by templates and we can use them to perform reduction on the graph.

REFERENCES

[1] R.J.M. Hughes, Super-combinators, *1982 ACM Symp. on LISP and Functional Programming*, 1-10, 1982.

[2] N. Jones and S. Mucknick, A Fixed Program Machine For Combinator Exspression Evaluation, *ACM Symp on LISP and Functional Programming*, 1982.

[3] A. Church, *The Calculi of Lambda Conversion*, Princeton University Press, 1941.

[4] N. Vardal, *Graph Reduction System Based on Combinator*, Msc. Thesis Baskent University Institute of Science, Ankara, 2001.

[5] J. B. Rosser, Highlights of the History of the Lambda calculus, *ACM on LISP and Functional Programming*, 1982.

[6] J. Fairbrain and S. C. Wray, TIM, *ACM third conference on Functional Programming Languages and computer Architecture*, New York ,1987.

[7] R. Milner, A Theory of Type Polymorphism in Programming, *J. Computer and System Science*, 3, 348-375, 1978.

[8] S. P. Jones, C. Clack, J. Salkild and M. Hardie, GRIP-A High Performance Architecture for Parallel Graph Reduction, *Third Conference on Functional Programming Languages and Computer Architecture*, 98-112, 1987.

[9] H. Abelson, G. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs*, McGraw Hill, New York, 1985.