On The Efficiency of Categorical Combinators as a Rewriting System

R. D. LINS
Computing Laboratory, The University of Kent, Canterbury, CT2 7NF, U.K.

SUMMARY
Categorical combinators form a formal system similar to Curry's combinatory logic. The original system was developed by Curien, inspired by the equivalence of the theories of typed lambda-calculus and Cartesian closed categories, as shown by Lambek and Scott. A new system for categorical combinators was introduced by the author. This system uses a more compact notation for the code and needs a smaller set of rewriting rules.

The aim of this paper is to analyse these two different rewriting systems for categorical combinators as a basis for implementation of applicative languages, and compare them with the classical approach due to Turner, using combinatory logic.

KEY WORDS Categorical combinators Turner's combinators Lambda calculus Functional programming
Complexity

INTRODUCTION
Categorical combinators form a formal system developed by Curien,¹ which is similar to combinatory logic. An approach for the execution of categorical combinators which uses a stack machine is described in Reference 5. As Turner showed,² stack machines do not provide an efficient implementation of lazy functional languages. Aiming to implement lazy functional languages in an efficient way using categorical combinators we developed a new system of categorical combinators called simplified categorical combinators.³ This system is based on the original system by Curien, but has the advantages of presenting a linear relationship between the size of a lambda-expression and its categorical combinator equivalent (the original system is quadratic in the worst case⁴), and working with a very simple and small set of rewriting rules to execute the code.

We shall compare the efficiency of these two categorical rewriting systems. A compiling-time optimization, the introduction of a fix-point combinator and some compiling time pre-processing techniques will be analysed.

We shall present a comparative analysis between the system of simplified categorical combinators and Turner's combinators as described in References 8 and 9, in terms of execution complexity in time and space.

¹ Permanent address: CHESF – Comp. Hidro Elétrica do São Francisco, R. Elpego Jorge de Souza, Bongi, 50.000 Recife – PE, Brazil.

0038–0644/87/080547–13$06.50
© 1987 by John Wiley & Sons, Ltd.

Received 25 April 1986
Revised 1 March 1987
The different machines will be implemented in Miranda\textsuperscript{,*} a polymorphically typed lazy functional language,\textsuperscript{9} supporting non-free algebraic data types, and thus forming an ideal experimental environment for work on reduction systems. The implementations will incorporate 'sharing' (of computation of duplicated sub-expressions) automatically.

THE ORIGINAL SYSTEM

Categorical combinators use DeBruijn's representation for variables. In DeBruijn\textsuperscript{10} notation for \( \lambda \)-calculus, a variable is replaced by a number corresponding to its position relative to the \( \lambda \) by which it is bound in the parse tree of the expression. The compilation algorithm for categorical combinators from DeBruijn's \( \lambda \)-calculus, as introduced by Curien,\textsuperscript{1} is

\[
\begin{align*}
\lambda \cdot a & \mapsto \Lambda([a]) \\
ab & \mapsto \text{App} \circ ([a], [b]) \\
[0] & \mapsto \text{Snd} \\
[n] & \mapsto \text{Snd} \circ \text{Fst}' \text{ if } n \geq 1, \text{ where Fst}^1 = \text{Fst} \text{ and Fst}^{n+1} = \text{Fst} \circ \text{Fst}'
\end{align*}
\]

In Reference 1 Curien chooses particular orientations of the axioms and deduces equations of a Cartesian closed category;\textsuperscript{2, 3} different selections of them will generate several different rewriting systems, for reducing the code generated by the compilation algorithm above. The system, which he calls CCL\textsubscript{eq}, uses the following laws, and simulates \( \lambda \)-calculus \( \beta \)-reduction by a sequence of elementary reduction steps of re-writings on the categorical code:

\[
\begin{align*}
(x \circ y) \circ z & \Rightarrow x \circ (y \circ z) \quad (r.1) \\
\text{Id} \circ x & \Rightarrow x \quad (r.2) \\
x \circ \text{Id} & \Rightarrow x \quad (r.3) \\
\text{Fst} \circ (x, y) & \Rightarrow x \quad (r.4) \\
\text{Snd} \circ (x, y) & \Rightarrow y \quad (r.5) \\
(x, y) \circ z & \Rightarrow (x \circ z, y \circ z) \quad (r.6) \\
\text{App} \circ (\Lambda(x), y) & \Rightarrow x \circ (\text{Id}, y) \quad (r.7) \\
\Lambda(x) \circ y & \Rightarrow \Lambda(x \circ (y \circ \text{Fst}, \text{Snd})) \quad (r.8)
\end{align*}
\]

The system is not locally confluent.\textsuperscript{1}

\textsuperscript{*} Trademark of Research Software Ltd.
ARITHMETIC OPERATORS

The way we work with constants plays a very important rôle in the efficiency of the system. In the original system of categorical combinators\(^1\) constants were introduced using a needlessly complicated notation. A constant \(k\) is represented in Reference 1 by

\[
k = \Lambda(Fst)k'
\]

The relationship between constants and functions over constants using composition led us to the inclusion of several rewriting laws.

We work with constants as described in Reference 4. A constant function with value \(k\) represents a whole class of arrows with the same name and codomain. This polymorphic way of introducing constants provided us with a very natural way of expressing the right composition of a constant arrow with any other arrow in the category by the addition of the following rule to our system

\[
c \circ x \Rightarrow c, \text{ where } c \text{ is a constant} \quad (r.9)
\]

Functions over constants are constant arrows themselves, which belong to this class. The interactions between constants and functions over constants are ruled by application. For example, the addition of two integers, say 2 and 3 will be represented by

\[
\text{App} \circ \text{App} \circ \langle \text{add}, 2 \rangle, 3 \Rightarrow 5
\]

where add is an arrow type \(1 \rightarrow [I \rightarrow I]\) and integer is of type \(1 \rightarrow I\) (\(I\) is the terminal object in the category and \(I\) is an object of the category). In reality arithmetic operations will take place outside the categorical world. We shall express numbers in general (integers, reals, etc) as constant arrows.

Each function over constants will have particular laws associated with it. For instance, the addition of a constant \(x\) to a constant \(y\) will be defined by the law

\[
\text{App} \circ <\text{App} \circ \langle \text{add}, x \rangle,y> \Rightarrow x + y
\]

We shall work with conditional expressions as being functions over constants. From now on when we refer to the original system of categorical combinations we shall be referring to the set of rewriting laws formed by rules (r.1) to (r.9).

SIMPLIFIED CATEGORICAL COMBINATORS

The compilation algorithm for simplified categorical combinators\(^4\) from DeBruijn \(\lambda\)-calculus is

\[
\begin{align*}
[\lambda.a] &\rightarrow \Lambda([a]) \\
[ab] &\rightarrow <[a],[b]> \\
[n] &\rightarrow n, \text{ where } n \text{ is a variable} \\
[c] &\rightarrow c, \text{ where } c \text{ is a constant}
\end{align*}
\]
This algorithm is based on the original one with some simplifications concerning the treatment of variables, constants and applications. The algorithm above presents a linear relationship between the size of source and compiled codes, whereas the original one, as was shown in Reference 6, is quadratic in the worst case.

The rewriting strategy we adopted in Reference 4 is leftmost-outermost, i.e. we shall look from left to right in an expression for the syntactically outermost pattern matching the left-hand side of any of the rewriting rules. When we find it this pattern will be rewritten and the rewriting will occur from the outermost level of the new expression. In Reference 7 we analysed the dynamics of the rewriting system of categorical combinators to avoid moving to the outermost level of an expression after each rewriting. The minimal set of rewriting rules to execute in a leftmost-outermost strategy the code generated by the compilation algorithm above is presented in Reference 4, as follows:

\[ n \circ (x,y) \Rightarrow (n-1) \circ x, \text{if } n > 0 \]  
\[ 0 \circ (x,y) \Rightarrow y \]  
\[ \langle x, y \rangle \circ z \Rightarrow \langle x \circ z, y \circ z \rangle \]  
\[ \langle \lambda(x), y \rangle \Rightarrow x \circ \langle \text{Id}, y \rangle \]  
\[ \langle \lambda(x) \circ y, z \rangle \Rightarrow x \circ \langle y, z \rangle \]  
\[ c \circ x \Rightarrow c, \text{where } c \text{ is a constant} \]

In Reference 4 we proved that leftmost-outermost reduction of simplified categorical combinators 'mimics' leftmost-outermost reduction of \( \lambda \)-terms, which is known to be a normalizing strategy.

IMPLEMENTATIONS OF THE CATEGORICAL MACHINES

Our implementations of the categorical machines are on top of Miranda, a polymorphically typed lazy functional language. Miranda is implemented by a graph reduction machine executing Turner's combinators. We shall need just one of its several features—algebraic data types. Using algebraic data types the implementation of algebraic systems consists only in declaring the constructors and the rewriting laws. The evaluation details of this nice feature of Miranda are explained in Reference 11. Figure 1 presents the core of the implementation of the (simplified set) categorical machine in Miranda.

The lines with the symbol `>` in the left margin are comment lines. The first non-comment line introduces the type of expressions (ex) with the declaration of the type constructors, pairing (P), composition (O), currification (L), application (A), identity (Id), variables (N) and constants (C). The other non-comment lines are the rewriting laws, as illustrated in the form (R.1) to (R.6) above, acting on items of that type. In order to have realistic programs in categorical combinators it is necessary to enrich the original set of rules to include rules for conditional expressions. We shall regard them as being functions over constants.
Simplified Categorical Combinator Machine on Turner's Miranda
\[\begin{align*}
\text{ex} &::= P \text{ ex ex} | O \text{ ex ex} | L \text{ ex} | A \text{ ex ex} | \text{Id} | \mathbb{N} \text{ num} | C \text{ cons} \\
0 &\times <x, y> \Rightarrow y \\
0 (N 0) (P x y) &\Rightarrow y \\
0 (n \times <x, y>) &\Rightarrow (n-1) \times x \\
0 (N n) (P x y) &\Rightarrow 0 (n (n-1)) x \\
0 (x, y) &\Rightarrow x + y \\
A \text{ x y} &\Rightarrow 0 x (P \text{ Id} y) \\
A (L(x), y) &\Rightarrow x + y \\
A (0 \times <x, y>) &\Rightarrow 0 \times (P y z) \\
C \times x &\Rightarrow C \\
0 (C x) y &\Rightarrow (C x)
\end{align*}\]

Figure 1.

The Miranda compiler provides us with three measures of complexity. Two of them refer to Miranda itself as an abstract machine, and are figures of space and time consumed in running a program. They are: the number of cells claimed, which corresponds to the number of Turner's combinators needed in the program, and the quantity of cycles, which is equal to the number of combinator reductions performed. The third measure refers to the c.p.u. time elapsed for running a program. This measure presents slight variations depending on the number of garbage collections performed during execution. We have here a conservative approach and will present always the worst value obtained. Each measure was read at least three times.

Our starting point in the comparison will be to look at test programs which are translated straight from $\lambda$-calculus to categorical combinators by the appropriate algorithm. These programs will be executed by abstract machines as similar as possible using algebraic data types. The closer the implementations are to each other the more reliable the information we get in terms of comparative complexity.

We must remark that the complexity measures provided by Miranda should by no means be interpreted as absolute values, but just as a fair comparison between the
complexities of the machines implemented, which must be properly interpreted and analysed.

THE TEST PROGRAMS

We chose three simple programs for which we were able to analyse the behaviour of the machines with respect to some of the most important features in applicative programming languages: the use of higher-order functions, recursion, and lazy evaluation. The programs are:

\[
\begin{align*}
\text{eval} & \ = \text{expression twice Succ 1, where} \\
\text{expression} & \ x \ y \ z \ = \ (((\text{x} \ x) \ y) \ y) \ z \\
\text{twice} & \ f \ x \ = \ f \ (f \ x) \\
\text{Succ} \ n & \ = \ (n + 1)
\end{align*}
\]

factorial \(n\), defined by the algorithm

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

and power \(x\ y\), defined by Pingala's algorithm,

\[
\begin{align*}
\text{power} \ x \ y & \ = \ 1, \text{ if } y = 0 \\
& \ = \ x, \text{ if } y = 1 \\
& \ = \ \text{power} \ (x \times x) \ (y/2), \text{ if } y \text{ is even} \\
& \ = \ x \times \ (\text{power} \ (x \times x) \ (y/2))
\end{align*}
\]

In these programs we make use of lazy evaluation in the implementation of conditional statements. The result of the evaluation of a conditional expression will yield an expression equivalent to the combinator \(K\), in case the conditional clause is true, or \(K I\) otherwise.

PERFORMANCE ANALYSIS

We will compare the performances of the rewriting machines formed by the original set of categorical combinators and the simplified one, using the test programs presented in the last section (see Table 1).

Since the simplified set presented a substantial performance difference, of at least an order of magnitude, showing a better behaviour as a rewriting system than the original set, we shall just analyse optimizations on top of this system.

INTRODUCING A FIX-POINT COMBINATOR

In the original set of categorical combinators there is no fix-point combinator. In the implementations discussed in the last section we used the compiled code for

\[
\emptyset = (\lambda a, \lambda b. b(aab))(\lambda c, \lambda d. d(ccd))
\]
to implement a fix-point operator.

The introduction of a primitive fix-point operator $Y$ in categorical combinators does not bring us any theoretical inconvenience. We shall introduce the fix-point combinator by the law

$$\langle Y, x \rangle \Rightarrow \langle x, \langle Y, x \rangle \rangle$$

In practical terms this will reduce the complexity of the categorical machine when recursion is necessary, as can be observed from Table II.
COMPLEMENTING $\lambda$-TERMS

We shall call an expression which is the categorical combinator translation of a $\lambda$-expression a $\lambda$-equivalent expression. An intermediate expression is a non $\lambda$-equivalent categorical combinator expression which we obtain by rewriting a $\lambda$-equivalent expression or another intermediate expression. We call a categorical combinator expression of the form $\Lambda(x)$, where $x$ is an arbitrary expression, a $\Lambda$-term. A complemented $\Lambda$-term is a $\Lambda$-term right composed with another expression forming an expression of the type $\Lambda(x) \circ y$. A highest level $\Lambda$-term is a $\Lambda$-term not enclosed within another $\Lambda$-term in an expression. We say that we Id-complement the highest level $\Lambda$-terms in an expression if whenever we find the outermost $\Lambda$-terms in an expression we Id-complement them, i.e. if we have an expression of type $\Lambda(x)$ we compose it with the Id combinator. For instance, the expression

$$\langle \Lambda(\lambda x. \Lambda(\langle 0,1\rangle)), \Lambda(0) \rangle$$

when Id-complemented will give us

$$\langle \Lambda(\lambda x. \Lambda(\langle 0,1\rangle)), \circ Id, \Lambda(0) \circ Id \rangle$$

As one can observe, the inner $\Lambda$-terms are not complemented.

If we analyse our set of rewriting laws we can see that (R.4) is a simplified case of (R.5):

$$\langle \lambda x. y \rangle \Rightarrow x \circ \langle Id,y \rangle$$
$$\langle \lambda x. y, z \rangle \Rightarrow x \circ \langle y, z \rangle$$

where the $\lambda$-term in the former is not Id-complemented. Our aim in this subsection is to prove that if we Id-complement all the highest-level $\Lambda$-terms we can delete (R.4) from our rewriting system.

We now analyse the cases where (R.4) is applicable and Id-complement all the necessary $\Lambda$-terms. If we have an expression type $\langle \Lambda(\lambda x), y \rangle$ it will be rewritten

$$\langle \langle \Lambda(\lambda x), y \rangle, z \rangle \Rightarrow \langle \Lambda(x) \circ \langle Id,y \rangle, z \rangle \quad (R.4)$$

The $\lambda$-term on the right-hand side of the expression above is already complemented. Thus we need not Id-complement it, because the pattern formed matches with the left-hand side of rule (R.5) already. But if we Id-complement the highest level $\lambda$-term of the expression on the left-hand side of the expression above we can use (R.5) instead of (R.4), thus:

$$\langle \langle \Lambda(\lambda x) \circ Id, y \rangle, z \rangle \Rightarrow \langle \Lambda(x) \circ \langle Id,y \rangle, z \rangle \quad (R.5)$$

So we can conclude that, whenever (R.4) is applicable, if we Id-complement just the highest level $\lambda$-terms, which form a pattern of type (R.4), rule (R.5) can be applied instead of (R.4). But just complementing the expressions of type (R.4) at compile time is not enough, as new (R.4) patterns are created during execution. This is the case for
expressions in which the right-hand side of an applicative pair reaches the left-hand side of another applicative pair during execution. Let us do an example. The expression \(((\lambda x \lambda y . x)(\lambda z . z))b)c\) will be translated into categorical combinators as
\[
\langle \langle \langle \langle \Lambda(1), \Lambda(0) \rangle, b \rangle, c \rangle
\]

If we \text{Id}-complement only the highest level \text{\lambda}-term in the \text{(R.4)} pattern we obtain
\[
\begin{align*}
\langle \langle \langle \langle \Lambda(1), \text{Id}, \Lambda(0) \rangle, b \rangle, c \rangle \\
\Rightarrow & \langle \langle \Lambda(1) \circ (\text{Id}, \Lambda(0)), b \rangle, c \rangle \\
\Rightarrow & \langle 0 \circ (\text{Id}, \Lambda(0)), b, c \rangle \\
\Rightarrow & \langle \Lambda(0), c \rangle
\end{align*}
\]

As we can see, an \text{(R.4)}-type expression was formed during execution, therefore it is necessary to \text{Id}-complement the expression \text{\lambda}(0) at compiling time as well, in order to remove this rule from our rewriting system.

We already analysed the possible interactions between the combinators \text{\langle}, \text{\rangle} (application) and \text{\Lambda}() (currying) and stated that patterns of type \text{\langle \Lambda(x), y \rangle} and \text{\langle x, \Lambda(y) \rangle} should be \text{Id}-complemented, if the subexpression \text{\Lambda(x)} is the highest-level \text{\lambda}-term, in order to delete rule \text{(R.4)} from our set of rewriting laws. Let us now analyse the interactions between the combinators \circ (composition) and \text{\Lambda}(). Expressions of type \text{\Lambda(x) \circ y} are already complemented \text{\lambda}-terms. A pattern of type \text{\langle x \circ \Lambda(y), x \rangle} will never occur, because it is not obtainable via compilation, and analysing our rewriting laws we can observe that subexpressions of type \text{\lambda(x)} will always appear on the left-hand side when interacting with compositions. Since we perform no associative rewrites, this relative position will remain unchanged. Analysing the way the ordinary pair combinator \text{\langle} is introduced in our code we can observe that a subexpression of type \text{\langle \Lambda(x), y \rangle} will never be generated. In patterns of type \text{\langle x, \Lambda(y) \rangle} the subexpression \text{\Lambda(y)} will only be rewriter if the pattern \text{\langle x, \Lambda(y) \rangle} is decomposed using rule \text{(R.2)} and the subexpression \text{\Lambda(y)} is embedded in an applicative context as analysed already. Because these are all the possible patterns in which an expression of type \text{\lambda(x)} can appear we can conclude that if we \text{Id}-complement all the highest level \text{\lambda}-terms of the expression we can delete rule \text{(R.4)} from our rewriting system.

As we can see in Table III, complementing the \text{\lambda}-terms brings us only a small improvement in performance. This is partly due to the necessity of complementing all outermost \text{\lambda}-terms.

\section*{Implementations of Turner's Machines}
Using the same structure of algebraic data types in Miranda we have implemented Turner's machines.

We implemented two versions of Turner's machines, those in References 6 and 8, respectively. The first one uses a smaller set of combinators, the second introduces three new combinators, i.e. B1, C1 and S1. The results of the execution of the same test programs are summarized in Table IV.
As we can see in Table IV the smaller set of Turner's combinators presents a better performance than the complete set. This contradicts the common views of functional programmers because it is a well-known fact that, on average, the complete set shows a better performance than the smaller set. The explanation is basically that, in the code of these programs the occurrence of S1, B1 and C1 combinators does not compensate the extra amount of pattern matching needed. Therefore we shall analyse the comparative behaviour of categorical combinators with the smaller set of Turner's combinators, which from now on we shall refer to as Turner's combinators. On the other hand, these figures are of the same order of magnitude.

COMPARATIVE ANALYSIS

We shall make a comparative analysis between the best results of Turner's and the simplified set with a fix-point of categorical combinator implementation, and comp-
Table V. Analysis of steady behaviour

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Turner's combinator classes</th>
<th>Categorical combinator classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>cycles</td>
<td>cells</td>
</tr>
<tr>
<td>power 7 0</td>
<td>1-141</td>
<td>1-878</td>
</tr>
<tr>
<td>power 7 1</td>
<td>2-346</td>
<td>4-102</td>
</tr>
<tr>
<td>power 7 10</td>
<td>15-254</td>
<td>27-785</td>
</tr>
<tr>
<td>power 7 20</td>
<td>19-017</td>
<td>34-489</td>
</tr>
<tr>
<td>power 7 40</td>
<td>22-414</td>
<td>40-854</td>
</tr>
<tr>
<td>power 7 60</td>
<td>23-390</td>
<td>42-712</td>
</tr>
<tr>
<td>power 7 80</td>
<td>25-811</td>
<td>47-219</td>
</tr>
<tr>
<td>power 7 100</td>
<td>26-299</td>
<td>48-148</td>
</tr>
<tr>
<td>power 7 120</td>
<td>26-821</td>
<td>49-112</td>
</tr>
<tr>
<td>power 7 140</td>
<td>29-730</td>
<td>54-548</td>
</tr>
<tr>
<td>power 7 160</td>
<td>29-242</td>
<td>53-619</td>
</tr>
<tr>
<td>power 7 180</td>
<td>30-218</td>
<td>55-477</td>
</tr>
<tr>
<td>power 7 200</td>
<td>29-730</td>
<td>54-548</td>
</tr>
<tr>
<td>power 7 220</td>
<td>30-706</td>
<td>54-406</td>
</tr>
<tr>
<td>power 7 240</td>
<td>30-218</td>
<td>55-477</td>
</tr>
<tr>
<td>power 7 260</td>
<td>32-639</td>
<td>59-984</td>
</tr>
<tr>
<td>power 7 280</td>
<td>33-127</td>
<td>60-913</td>
</tr>
<tr>
<td>power 7 300</td>
<td>33-615</td>
<td>61-842</td>
</tr>
</tbody>
</table>

implemented A-terms. In order to analyse the stability of the data obtained we attribute a series of different values to the factorial and power programs, and the results are presented in Tables V and VI.

One can observe the stability of the results obtained and that we have basically the same ratio of complexity between the two machines as shown formerly. The data presented in the tables above show that in large programs the system of simplified categorical combinator tends to take advantage of its compact code and to present a better performance than Turner's combinator. The previous results we obtained are summarized in Table VII.

CONCLUSIONS

The experiments presented in this paper allow us to state that the system of simplified categorical combinator presents a performance of at least an order of magnitude better than the original system of categorical combinator, and also time and space complexity of the same order of magnitude as Turner's combinators.

ACKNOWLEDGEMENTS

The author would like to express his gratitude to Prof. David Turner and Dr. Simon Thompson for their advice, suggestions and comments.

The financial support for this work was provided by CNPq, Brazil, Grant 20. 2744/84.
Table VI. Analysis of steady behaviour

<table>
<thead>
<tr>
<th>Program</th>
<th>Turner’s combinator cycles</th>
<th>cells</th>
<th>c.p.u.</th>
<th>Categorical combinator cycles</th>
<th>cells</th>
<th>c.p.u.</th>
</tr>
</thead>
<tbody>
<tr>
<td>factorial 0</td>
<td>981</td>
<td>1-598</td>
<td>0-43</td>
<td>851</td>
<td>1-212</td>
<td>0-33</td>
</tr>
<tr>
<td>factorial 1</td>
<td>2-659</td>
<td>4-740</td>
<td>1-25</td>
<td>1-567</td>
<td>2-487</td>
<td>0-63</td>
</tr>
<tr>
<td>factorial 10</td>
<td>12-772</td>
<td>24-165</td>
<td>6-73</td>
<td>6-832</td>
<td>11-757</td>
<td>3-55</td>
</tr>
<tr>
<td>factorial 20</td>
<td>24-192</td>
<td>45-914</td>
<td>15-03</td>
<td>12-860</td>
<td>22-212</td>
<td>6-62</td>
</tr>
<tr>
<td>factorial 30</td>
<td>35-212</td>
<td>67-284</td>
<td>22-30</td>
<td>18-480</td>
<td>32-272</td>
<td>9-78</td>
</tr>
<tr>
<td>factorial 40</td>
<td>46-232</td>
<td>88-654</td>
<td>27-85</td>
<td>24-100</td>
<td>42-332</td>
<td>13-70</td>
</tr>
<tr>
<td>factorial 50</td>
<td>57-252</td>
<td>110-024</td>
<td>33-35</td>
<td>29-720</td>
<td>52-392</td>
<td>17-85</td>
</tr>
<tr>
<td>factorial 60</td>
<td>68-272</td>
<td>131-394</td>
<td>44-73</td>
<td>35-340</td>
<td>62-452</td>
<td>24-03</td>
</tr>
<tr>
<td>factorial 70</td>
<td>79-323</td>
<td>152-793</td>
<td>49-87</td>
<td>40-993</td>
<td>72-545</td>
<td>26-85</td>
</tr>
<tr>
<td>factorial 80</td>
<td>90-343</td>
<td>174-163</td>
<td>55-17</td>
<td>46-613</td>
<td>82-605</td>
<td>29-53</td>
</tr>
<tr>
<td>factorial 90</td>
<td>101-363</td>
<td>195-533</td>
<td>60-92</td>
<td>52-233</td>
<td>92-665</td>
<td>31-87</td>
</tr>
<tr>
<td>factorial 100</td>
<td>112-383</td>
<td>216-903</td>
<td>66-43</td>
<td>57-853</td>
<td>102-725</td>
<td>34-73</td>
</tr>
<tr>
<td>factorial 110</td>
<td>123-403</td>
<td>238-273</td>
<td>81-03</td>
<td>63-473</td>
<td>112-785</td>
<td>37-40</td>
</tr>
<tr>
<td>factorial 120</td>
<td>134-423</td>
<td>259-643</td>
<td>86-43</td>
<td>69-093</td>
<td>122-845</td>
<td>40-07</td>
</tr>
<tr>
<td>factorial 130</td>
<td>145-433</td>
<td>281-013</td>
<td>92-13</td>
<td>74-713</td>
<td>132-905</td>
<td>42-66</td>
</tr>
<tr>
<td>factorial 140</td>
<td>156-463</td>
<td>302-383</td>
<td>108-88</td>
<td>80-333</td>
<td>142-965</td>
<td>50-40</td>
</tr>
<tr>
<td>factorial 150</td>
<td>167-483</td>
<td>323-753</td>
<td>114-38</td>
<td>85-953</td>
<td>153-025</td>
<td>53-05</td>
</tr>
</tbody>
</table>

Table VII

<table>
<thead>
<tr>
<th>Program</th>
<th>twice twice twice succ 1 cycles</th>
<th>cells</th>
<th>c.p.u.</th>
<th>factorial 5 cycles</th>
<th>cells</th>
<th>c.p.u.</th>
<th>power 5 7 cycles</th>
<th>cells</th>
<th>c.p.u.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Turner’s combinator</td>
<td>8-487</td>
<td>15-961</td>
<td>4-05</td>
<td>7-128</td>
<td>13-344</td>
<td>3-42</td>
<td>12-158</td>
<td>22-119</td>
<td>6-37</td>
</tr>
<tr>
<td>Categorical combinator</td>
<td>4-721</td>
<td>7-993</td>
<td>2-07</td>
<td>3-880</td>
<td>6-575</td>
<td>1-70</td>
<td>4-546</td>
<td>7-454</td>
<td>2-00</td>
</tr>
</tbody>
</table>

REFERENCES


