Statistics on Graph Reduction of SASL Programs

PIETER H. HARTEL
Computing Science Department, University of Amsterdam, Nieuwe Achtergracht 166,
1018 WV Amsterdam, Netherlands

AND

ARTHUR H. VEEN
Computing Science Department, University of Nijmegen, Toernooiveld 1,
6525 ED Nijmegen, Netherlands

SUMMARY

The execution has been studied of four small and four medium-sized SASL programs, when interpreted by a variant of Turner's combinator reducer. Size, structure and composition of the combinator graph have been analysed at frequent intervals during the reduction process. The most interesting results are summarized and discussed. Nodes of the graph live rather short lives and are usually not shared. Cycles are rare, and linear lists are often short. In most aspects the behaviour of the graph is quite ordinary in the sense that a simple model is sufficient to obtain a good approximation.

KEY WORDS Combinators Graph reduction Statistical model

INTRODUCTION

In our project to design a parallel reduction machine for the efficient execution of functional programs,¹ we have become increasingly aware that the architecture should not just be language-based, i.e. exploit properties of the language for which it is designed, but also application based, i.e. exploit properties of the class of applications for which it is likely to be used.² However, the experience with functional languages is so limited that the nature, let alone the magnitude, of such properties is unclear. In the absence of definite data the architect often depends on intuition, but the intuition that we as programmers or as language implementers have developed in a mostly imperative world is a dangerous guide when it concerns the execution of functional languages implemented with lazy evaluation.

Already in our preliminary design effort we encountered the need for statistics about program properties. Most implementers of functional languages would agree that graph

¹ Current affiliation: Computing Science Department, University of Amsterdam.
reduction, which evaluates shared subexpressions only once, is much more efficient than the conceptually simpler string reduction. One of the architectural options we were considering at the time was to forego locally the savings of graph reduction to reduce communication costs. In our analysis of this trade-off we needed to know the frequency and the distribution of shared subexpressions in a typical application program. We were surprised to find that the literature did not provide any such statistic.

Delimiting a sensible class of applications for which to design our machine is a difficult issue which we have not yet resolved to our satisfaction. In this paper we sidestep the issue by choosing a few SASL programs that we assume to belong to that class. Statistics will be presented that have been gathered from these applications by an interpreter especially instrumented to monitor the reduction process and to analyse the graph at regular intervals. Below a motivation is given for the choices that have been made concerning applications, programming language and method of translation. The methodology and the statistical results of the experiment are presented in subsequent sections.

Scope of the study

In the literature many functional languages have been reported, but very few sizeable applications. Most of these are written in SASL and since we have access to several of these programs as well as an implementation that could be easily instrumented to collect statistics, we have limited our study to SASL programs. We do not expect that the statistics reported here would be significantly different for other functional languages with lazy semantics, if implemented in a similar fashion.

The standard implementation of SASL is the combinator implementation described by Turner. This implementation is the most suitable one for our purpose, since it contains very few optimizations that could bias our statistics. We used a locally produced variation, which employs the same set of combinators and abstraction rules and behaves identically to Turner’s implementation with respect to the statistics reported in this paper. We repeated all measurements with another implementation that reduces the number of I-reductions by using a somewhat different allocation strategy and adds special combinators as debugging aids. This experiment was reassuring in the sense that all differences in the statistics could easily be related to the known differences in the implementations and none of the conclusions drawn in this paper were invalidated.

It is well known that the fixed set of simple combinators Turner uses leads to considerable overhead. Spectacular improvements have been reported either by using program-derived combinators or by avoiding graph transformations for those parts of the computation that can be performed eagerly. We certainly plan to incorporate some of these improvements, or variations thereof, in our design, but which ones is not yet clear: the choice depends in part on the statistics reported here. For such improved combinator reducers many of our statistics would be quite different. However, since all combinator-based reducers reported so far can be considered to be optimizations of Turner’s implementation, our statistics should still be useful as a basis for extrapolation or as a yardstick to measure progress.

The choice of the application programs has been ad hoc. We are only interested in an application program as far as it can be considered to be representative of a large class of applications. In our choice we are trying to approximate the average and avoid the unusual. We have a few dozen SASL programs at our disposal, most of them
small. Toy programs can usually be made to consume a considerable amount of computing time by providing them with appropriate input data, but we feel that this kind of extrapolation is a dangerous practice. Realistic programs not only represent a sizeable computation, but are also large in terms of program text. We are therefore mainly interested in larger programs. A few toy programs were included in our test set in an attempt to quantify our objection. Because the SASL interpreter that has been instrumented is not a particularly efficient one, collecting the statistics with a reasonable accuracy is a time-consuming process. Therefore only executions of at most a few hundred thousand reductions could be handled. This is equivalent to less than a minute CPU time on some of the most efficient implementations. We had access to four programs of about the right size. Three of them had the extra advantage that they have been described in the literature.

**Related literature**

In the past the study of compilation schemes and the composition and complexity of combinatory code has been given some attention. Almost always, worst-case complexity analysis is used, but some studies\(^7\), \(^8\) analyse the average case defined in a theoretical way by averaging over all possible expressions and assuming that all expressions are equally likely to occur. We have studied the average case in a practical sense, i.e. based on statistics gathered from programs occurring in practice.

The literature contains several sources of statistical data on programs, but none of the figures is directly applicable to graph reduction. Many data have been collected for imperative languages implemented on a conventional machine; Weicker\(^9\) compiled a list of 16 collections on languages such as Fortran, PL/1, Algol68, Pascal and Ada. Because functional languages supposedly lead to a radically different programming style, their figures are not useful to us, with the possible exception of the relative frequency of arithmetic and comparative operations.

Studies of LISP programs may be slightly more useful, because LISP, although it contains imperative constructs, also has function application as its main construction device and binary lists as its main data structure. Gabriel\(^10\) uses various bench-marks some of which could be considered as typical programs of a particular artificial intelligence area (such as theorem proving, expert systems, game playing etc.). The focus of the book is, however, on the implementation, and consequently statistics of typical programs can only be inferred indirectly. A wealth of statistics on the allocation of lists by an Interlisp-10 implementation has been collected by Clark and Green.\(^11\) Their focus has been on opportunities for space-efficient encoding, a much more specific concern than ours.

Peyton Jones compares the efficiency of combinator reduction according to Turner to that of applicative order and normal order $\lambda$-reduction.\(^12\) The experimental results with a set of small test programs indicate that combinator reduction can be efficiently implemented.

**METHODOLOGY**

We have run a set of eight existing SASL programs on an interpreter that we had instrumented with a data-collection procedure.
The SASL graph reduction system

Our implementation of SASL closely resembles Turner's implementation.\textsuperscript{3} It consists of a compiler to translate a source-language program into an abstract syntax graph and an evaluator which transforms this representation of the program, step by step, into the representation of its result. A representation has the form of a rooted directed binary graph. Interior nodes represent function applications and leaves represent primitive values or functions. The left descendant of an interior node stands for the function and the right descendant for the argument to which it is applied.

The evaluator is capable of applying about three dozen different transformations to a program graph. Each transformation (reduction step) corresponds to the firing of a primitive function. In addition to the 'standard' combinators and arithmetic/logic functions,\textsuperscript{3,13} Turner uses combinators to support pattern matching on arguments of SASL functions (TRY and MATCH) and various optimizations of the standard combinators (e.g. Sp is shorthand for (S' P) and Us is a strict version of U). In its main loop, the evaluator searches the graph leftmost depth-first for a primitive function supplied with a sufficient number of arguments. If such a function is found, the corresponding transformation is applied to the graph and the search continues. Evaluation terminates when no more reducible expressions can be found.

Two issues related to the implementation of storage allocation that Turner could afford to ignore in his paper need to be described here:

1. For all constants of type boolean, character, nil or combinator, one node is permanently allocated to represent its value. Such leaves are not included in any of the statistics, because they would bias the values for sharing of nodes.
2. The P combinator is used so often (in list construction) that Turner has introduced a special interior node to be used instead of a Curried application of P to two arguments (see Figure 1). Selecting the n'th element of a list is executed as n successive reduction steps.

Set of application programs

The test set consists of eight different programs. Four programs are small and are run with a small input data set. The four other programs are all of medium size. They are run on small input data sets. The following list provides a short description of each program, the input to which it was applied and the motivation for including it in our test set.

![Diagram of a combinator tree](image)

*Figure 1. Optimization of storage allocation for list construction. A node marked '@' indicates an application node and ':' a constructor node*
fib 7
prints the seventh Fibonacci number using double recursion.
This function appears as an example in most texts on functional programming. Its
behaviour during graph reduction may therefore be of general interest.

qsort (9, 0, 1, 8, 4, 8)
sorts a list of 6 numbers using Quicksort.
Apart from being another standard example, this algorithm is typical for the divide
and conquer approach to program development, which we expect to become very
important for parallel evaluation.

hamming 100
prints in ascending order the first 100 natural numbers whose prime factors are 2, 3
and 5 only.\textsuperscript{14}
The interesting aspect of this solution is that the output list is shared among many
computations via cycles in the representation. In an implementation that does not
support cycles this solution cannot be executed efficiently.

paraff 5
enumerates in order of increasing size the first five paraffin molecules, similar to
Turner's original program in KRC.\textsuperscript{15}

wave 5
a simplified version of a program that predicts the tides in the North Sea. It assumes
a square region with constant depth and simulates five discrete time steps of 20
minutes.
This program is the subject of a study towards developing functional programs
that are suitable for parallel evaluation on coarse grain parallel architectures.\textsuperscript{16}

em
implements a subset of the commands that are supported by the standard UNIX
editor ed.
Its input script consists of a call to the help command, three calls to the command
to read in a file and a call to the command to print the entire contents of the edit
buffer. The input file is the same as that prepared for the yacc program (see below). The em program has been developed as an exercise in functional programming for a
problem area where functional programming is often claimed to be unsatisfactory.\textsuperscript{17}

lambda
an implementation of the $\lambda$-K calculus,\textsuperscript{18} with some added delta rules to support
arbitrary-precision arithmetic.
This program was written by one of the authors. The input data consist of the
definitions of the standard combinators S and K in the $\lambda$-calculus and the application
(S K K).

yacc
essentially a rewritten version of the UNIX parser generator of the same name.
It was developed with roughly the same motivation as the em program.\textsuperscript{19} The
input data to the yacc program consist of a description of the syntax of input to
yacc itself. The output produced is a complete parser of yacc input, written in
SASL.

Table I gives an indication of size and complexity of the eight programs that were
selected. The number of lines of pure program text (i.e. excluding comments and
blank lines) is a measure of program size. The number of functions defined at global
level (i.e. excluding WHERE definitions) is provided as a measure of the program
structure. The total number of (combinator) reduction steps is considered to be representative for the computational complexity of a program. On the average about one node is claimed per reduction step.

Data collection method

The aim of the experiment is to determine the numerical values of a number of parameters that characterize both the process of normal-order graph reduction (e.g. the average number of nodes claimed per reduction step) as well as the intermediate graphs that occur during reduction (e.g. the number of nodes in the graph). For some parameters we are also interested in their fluctuations in time. Although some reduction steps, such as the TRY combinator, take longer to execute than others, we suppose this variation to be unrelated to any of the factors being studied and count time in reduction steps. Each reduction step (i.e. the graph rewrite due to a single combinator) is considered to consume one unit of time. After a certain number of reduction steps (the sampling interval) the evaluator passes control to the data-collection procedure, which traverses the complete graph.

We assume that the sampling is unbiased, i.e. that none of the programs that are studied contains periodic phenomena with a period related to the chosen sampling interval. This implies that the reliability of the statistics we quote is only dependent on the number of samples. The sample intervals chosen for the test programs and the resulting number of samples are shown in Table II.

We have repeated all measurements with a sample interval approximately ten times larger and found only a significant difference in the measurements of the reference count of the current redex (see below).

<table>
<thead>
<tr>
<th>Sample interval</th>
<th>1</th>
<th>1</th>
<th>7</th>
<th>11</th>
<th>211</th>
<th>211</th>
<th>67</th>
<th>401</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of samples</td>
<td>233</td>
<td>712</td>
<td>1855</td>
<td>2264</td>
<td>1979</td>
<td>1178</td>
<td>1348</td>
<td>1698</td>
</tr>
</tbody>
</table>

RESULTS

In the following paragraphs, we present the statistics that we have gathered.
Instruction mix

The majority of the combinators executed by a graph reducer based on Turner’s combinators serve to place arguments in the right position. This is a major source of inefficiency,\(^4\)\(^,\)\(^6\) as can be seen from the data in Table III. In the case of the yacc program, for instance, no more than 5 per cent of all combinators executed perform ‘real work’ (HD, TL, list selection, APPEND and the arithmetic and logic operators). Particularly worrying is the popularity of the identity function I. On the positive side is the success of the optimization that uses the B and C combinators and their derivatives instead of the S combinator.

<table>
<thead>
<tr>
<th>Table III. Percentages of combinators executed</th>
</tr>
</thead>
</table>
| \(\begin{array}{cccccccc}
| & \text{fib} & \text{sort} & \text{hamming} & \text{paraff} & \text{wave} & \text{em} & \text{lambda} & \text{yacc} \\
| \text{HD, TL, select} & 0 & 2.1 & 0 & 0.5 & 16.3 & 1.3 & 2.9 & 3.2 \\
| \text{APPEND} & 0 & 2 & 0 & 0 & 0 & 15.8 & 0.1 & 0.1 \\
| \text{Operators} & 15.5 & 2.1 & 4.9 & 5.8 & 12.6 & 9.7 & 3.4 & 1.9 \\
| I & 17.7 & 9.9 & 24.2 & 23.1 & 25.8 & 6.7 & 47.3 & 19.7 \\
| Y & 0 & 0.7 & 0 & 0.8 & 0.3 & 0.2 & 0.7 & 0.8 \\
| K & 0 & 0.7 & 0 & 0.8 & 0.3 & 0.2 & 0.7 & 0.8 \\
| S, S', Sp & 15.1 & 1 & 5.8 & 8.1 & 6.7 & 0.5 & 5.9 & 3.8 \\
| B, B', Bp & 34.5 & 22.3 & 20.6 & 16.7 & 19.8 & 17.2 & 34.7 & 42.7 \\
| C, C', Cp & 11.4 & 8.1 & 10.7 & 12.4 & 3.6 & 6.3 & 7.6 & 6.3 \\
| U, Us & 0 & 5.2 & 5.2 & 4.3 & 3.0 & 11.5 & 4.2 & 8.7 \\
| COND & 0 & 2.6 & 2.6 & 2.8 & 0.8 & 0.2 & 1.1 & 1.1 \\
| TRY & 21.6 & 3.9 & 6.0 & 8.8 & 0.5 & 10.6 & 3.1 & 1.7 \\
| MATCH & 19.4 & 3.9 & 6.0 & 7.7 & 0.5 & 10.5 & 2.2 & 2.3 \\
| \end{array}\) |

The size and growth of a graph

The number of nodes in the program graph as a function of time gives an indication of the amount of space required. In particular, if this function fluctuates wildly, the demands on the storage allocator are distributed very unevenly in time. In order to avoid long disruptive pauses caused by the garbage collector, special measures may be considered.\(^3\)

The size of the graph for the medium-sized programs is depicted in Figure 2. The horizontal axis represents normalized time expressed as proportion of the total number of reduction steps for the entire program; the vertical axis shows the number of nodes as a proportion of the maximum attained (shown at top). The results indicate that the total number of nodes in a graph changes rather gradually. All major transitions in the size of the graph can be related easily to the algorithm. The yacc and lambda programs are similar in the sense that the graph builds up rapidly and then enters a relatively steady state until near the end. The behaviour of the wave program (and to a lesser extent that of the em program) is interesting because it shows one of the drawbacks of lazy semantics. For more than 95 per cent of its execution time it is building up a large graph that reflects the propagation of the demand to the initial boundary conditions. Most of the real work occurs when this graph is reduced near the end of the computation. We expect this phenomenon to become less pronounced when the simulation is extended over more time steps.
The shape of a graph

An important parameter for a graph-traversal algorithm is the shape of the average graph. For a recursive descent algorithm, for instance, a balanced graph uses less stack space than a graph of the same size that is severely out of balance. To obtain a measure for this aspect we recorded the depth of each leaf node during leftmost depth-first traversal. The average depth is of course dependent on the size of the graph. We would like to have a more scale-invariant measure. We know that the average depth of planted plane trees is approximately $\sqrt{n}$, where $n$ is the number of nodes. Hence, if our graphs behave like the average tree, the relative depth of a node in a graph, defined as

$$\text{relative depth} = \frac{\text{depth}}{\sqrt{\text{number of nodes}}}$$

would be reasonably scale-invariant. We have recorded the relative depth of each node during each traversal (see Table IV).
Table IV. Ratio between average depth and square root of the number of nodes

<table>
<thead>
<tr>
<th></th>
<th>fib</th>
<th>qsort</th>
<th>hamming</th>
<th>paroff</th>
<th>wave</th>
<th>em</th>
<th>lambda</th>
<th>yacc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>1.6</td>
<td>1.5</td>
<td>2.3</td>
<td>1.3</td>
<td>1.2</td>
<td>0.7</td>
<td>0.9</td>
<td>2.0</td>
</tr>
<tr>
<td>95th percentile</td>
<td>2.4</td>
<td>2.4</td>
<td>4.6</td>
<td>2.3</td>
<td>2.0</td>
<td>1.1</td>
<td>1.5</td>
<td>9.4</td>
</tr>
</tbody>
</table>

The composition of a graph

During the evaluation of the eight programs, the composition of their graphs was recorded. In Table V the average percentages of structure nodes are listed. The standard deviation of these figures is less than 10 per cent of the average.

The figures indicate that there is certainly an advantage in the use of constructor nodes. For instance, without the constructor nodes 26 per cent more nodes would have been required for the *em* program. When interpreting these figures, remember that leaves that represent constants of type boolean, character, nil or combinator are not counted as part of the graph.

The structure of a graph

In the combinator code that is evaluated by the SASL interpreter, there are two kinds of simple subgraphs visible that may be replaced by single nodes: constructor nodes chained via their tail fields and application nodes chained via their head fields. The former may be interpreted as the representation of a completely linear list and the latter as the application of a single function to multiple arguments. During the experiments, occurrences of both have been recorded by counting the lengths of the respective chains. Applications of the identity function were ignored. A summary of these figures is shown in Table VI. It shows for instance that for our four medium-sized programs at least 90 per cent of all functions have one or two arguments. In an implementation based on supercombinators these figures would probably be significantly higher. All data structure in SASL programs must be built out of lists. In a language which provides additional data structuring tools, such as records or tuples, fewer short lists would be used than is the case here.

The use of sharing

The major advantage of graph reduction over string reduction is the possibility to exploit sharing. Sharing can be beneficial to avoid duplication of work and duplication of (program or data) storage. The use of sharing has been investigated from a number of different perspectives.

Table V. Average composition (in per cent) of SASL graphs

<table>
<thead>
<tr>
<th></th>
<th>fib</th>
<th>qsort</th>
<th>hamming</th>
<th>paroff</th>
<th>wave</th>
<th>em</th>
<th>lambda</th>
<th>yacc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>78</td>
<td>92</td>
<td>64</td>
<td>80</td>
<td>83</td>
<td>73</td>
<td>86</td>
<td>83</td>
</tr>
<tr>
<td>Constructor</td>
<td>–</td>
<td>4</td>
<td>20</td>
<td>17</td>
<td>11</td>
<td>26</td>
<td>11</td>
<td>15</td>
</tr>
</tbody>
</table>
Table VI. Distribution of list lengths and argument counts

<table>
<thead>
<tr>
<th>List length</th>
<th>fib</th>
<th>qsort</th>
<th>hamming</th>
<th>paraff</th>
<th>wave</th>
<th>en</th>
<th>lambda</th>
<th>yacc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 element</td>
<td>62%</td>
<td>41%</td>
<td>12%</td>
<td>5%</td>
<td>17%</td>
<td>43%</td>
<td>21%</td>
<td></td>
</tr>
<tr>
<td>2 elements</td>
<td>2%</td>
<td>0.4%</td>
<td>43%</td>
<td>9%</td>
<td>74%</td>
<td>19%</td>
<td>32%</td>
<td></td>
</tr>
<tr>
<td>3 or 4 elements</td>
<td>14%</td>
<td>0.9%</td>
<td>33%</td>
<td>8%</td>
<td>0.3%</td>
<td>20%</td>
<td>37%</td>
<td></td>
</tr>
<tr>
<td>5-8 elements</td>
<td>22%</td>
<td>2%</td>
<td>11%</td>
<td>17%</td>
<td>0.5%</td>
<td>6%</td>
<td>5%</td>
<td></td>
</tr>
<tr>
<td>95th percentile</td>
<td>5</td>
<td>92</td>
<td>6</td>
<td>12</td>
<td>26</td>
<td>14</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument count</th>
<th>1 argument</th>
<th>2 arguments</th>
<th>95th percentile</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0%</td>
<td>80%</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>44%</td>
<td>40%</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>39%</td>
<td>37%</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>28%</td>
<td>59%</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>13%</td>
<td>76%</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>54%</td>
<td>39%</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>41%</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>66%</td>
<td>29%</td>
<td>2</td>
</tr>
</tbody>
</table>

Reference counting

An important parameter for the storage allocation and reclamation system is the number of pointers to a node. For instance in a system which uses reference counting it is important to know how often the reference count field overflows. The reference count taken over all nodes in all sample graphs has been measured. The average values and 95th percentiles are shown in the first two rows of Table VII. The distribution is such that for the medium-sized programs less than 10 per cent of all nodes are shared (third row). The histograms of reference counts over nodes for the four medium-sized programs are plotted in Figure 3. The horizontal axis represents the reference count and the vertical axis the proportion of the nodes with such a reference count.

During normal-order graph reduction, there is always a current redex. This redex can be identified with an application node in the graph. The bottom half of Table VII shows the sharing figures on the current redex node. Apparently shared nodes have a higher probability of being selected as current node. These figures may not be construed to imply that graph reduction is on the average twice as efficient as string reduction. In string reduction the S combinator, for instance, would copy its argument graph once, but the contractum of the S reduction may appear later as an argument to another S reduction, such that the number of copies rises exponentially. We have run some of our programs in string reduction mode, generally on smaller input data sets, and found the number of reduction steps to be larger by orders of magnitude.

When we repeated the measurements with a larger sample interval, the figures for sharing of the current redex in the en program were reduced by a factor of 2. We could not find an explanation for this discrepancy.

Cycles

In a storage allocation system that employs reference counting the presence of cycles in the graph requires special attention. Information about the occurrences of cycles and their length may be useful to delimit the scope of the problem. The data collection algorithm visits the nodes of the graph leftmost depth-first, such that each edge is traversed exactly once. A cycle is counted whenever an edge joins the current path from the root of the graph. The experimental data (Table VIII) show that cycles are
small but occur relatively frequently. In the medium-sized programs, there is on the average one cycle per 200 nodes, with an average length of 15 nodes. All programs are recursive, which in this implementation gives rise to cycles (the knot-tying Ψ combinator). In the hamming, paraff and wave programs cyclic data structures also occur.

Ongoing research at our institute is investigating the question of how cycles can be avoided. As an illustration we present some preliminary results in the lower half of Table VIII. These were produced by replacing the knot-tying Ψ combinator in our reducer by one that avoids cycles. This leads to a large increase in the number of reduction steps. To limit this increase we modified seven of our SASL programs by lifting all recursive WHERE clauses to global level and removing WHERE clauses with multiple definitions. Recursive global functions still require pointers back to their roots, but these were threaded through an indirection table and not counted as cyclic.
Table VII. Reference counts

<table>
<thead>
<tr>
<th></th>
<th>fib</th>
<th>qsort</th>
<th>hamming</th>
<th>paraff</th>
<th>wave</th>
<th>em</th>
<th>lambda</th>
<th>yacc</th>
</tr>
</thead>
<tbody>
<tr>
<td>All nodes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>average</td>
<td>1.17</td>
<td>1.13</td>
<td>1.13</td>
<td>1.21</td>
<td>1.50</td>
<td>1.17</td>
<td>1.14</td>
<td>1.16</td>
</tr>
<tr>
<td>95th percentile</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>percentage shared</td>
<td>13</td>
<td>8</td>
<td>11</td>
<td>14</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Current redex</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>average</td>
<td>1.9</td>
<td>1.3</td>
<td>1.5</td>
<td>2.1</td>
<td>2.1</td>
<td>2.9</td>
<td>1.7</td>
<td>2.1</td>
</tr>
<tr>
<td>95th percentile</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>22</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>percentage shared</td>
<td>56</td>
<td>19</td>
<td>37</td>
<td>47</td>
<td>36</td>
<td>20</td>
<td>56</td>
<td>34</td>
</tr>
</tbody>
</table>

Table VIII. Cycles

<table>
<thead>
<tr>
<th></th>
<th>fib</th>
<th>qsort</th>
<th>hamming</th>
<th>paraff</th>
<th>wave</th>
<th>em</th>
<th>lambda</th>
<th>yacc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard cyclic implementation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cycles per 1000 nodes</td>
<td>30.1</td>
<td>23.1</td>
<td>17.5</td>
<td>9.3</td>
<td>0.7</td>
<td>2.3</td>
<td>10.9</td>
<td>4.8</td>
</tr>
<tr>
<td>average length</td>
<td>4.8</td>
<td>7.0</td>
<td>15.4</td>
<td>13.2</td>
<td>17.1</td>
<td>7.8</td>
<td>12.9</td>
<td>22.7</td>
</tr>
<tr>
<td>Cycle avoiding experiment</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cycles per 1000 nodes</td>
<td>0</td>
<td>0</td>
<td>3.2</td>
<td>0.0009</td>
<td>0.002</td>
<td>0</td>
<td>0</td>
<td>not run</td>
</tr>
<tr>
<td>average length</td>
<td>0</td>
<td>0</td>
<td>54.4</td>
<td>31</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>not run</td>
</tr>
<tr>
<td>extra reduction steps</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0.16%</td>
<td>7.4%</td>
<td>7.3%</td>
<td>9.8%</td>
<td>not run</td>
</tr>
</tbody>
</table>

The life span of nodes

The life of a node starts when it is claimed by the evaluator and ends when it becomes unreachable from the root. Information about the life span of nodes and its distribution could be used for instance to design a hierarchical node store, where the residence of a node is determined by its retention period, as exploited by Ungar.20

In our experiments we determined an upper bound for the life expectancy of nodes: the creation time of each node is known exactly, but the expiration of nodes that are part of a cycle is assumed to have occurred at the time at which the data collection algorithm discovered the node to be garbage (Table IX). Because cycles are rare this lack of precision is only slight.

From the experiments it was found that most nodes have a short life: about 60 percent of the nodes witness no more than 10 reduction steps. The histograms obtained for the medium-sized programs are shown in Figure 4. The horizontal axis represents lifetime and the vertical axis the proportion of the nodes with a lifetime around that value.
Figure 4. Histograms of the life time of nodes in reduction steps

Table IX. Life span of nodes measured in reduction steps (per cent)

<table>
<thead>
<tr>
<th></th>
<th>fib</th>
<th>qsort</th>
<th>hamming</th>
<th>paraff</th>
<th>wave</th>
<th>em</th>
<th>lambda</th>
<th>yacc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 step</td>
<td>1</td>
<td>15</td>
<td>21</td>
<td>15</td>
<td>10</td>
<td>28</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>2–10 steps</td>
<td>68</td>
<td>48</td>
<td>43</td>
<td>41</td>
<td>55</td>
<td>29</td>
<td>44</td>
<td>54</td>
</tr>
<tr>
<td>11–100 steps</td>
<td>20</td>
<td>18</td>
<td>25</td>
<td>19</td>
<td>15</td>
<td>23</td>
<td>19</td>
<td>16</td>
</tr>
<tr>
<td>100–2500 steps</td>
<td>11</td>
<td>19</td>
<td>9</td>
<td>17</td>
<td>10</td>
<td>14</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>
CONCLUSIONS

The most notable property of the graphs we have measured is their lack of distinction. They do not have disproportionally long paths, but are not very balanced either. The graphs do not grow or shrink in sudden bursts. In most aspects their structure as well as their behaviour is quite ordinary. This is fortunate, because it means that the behaviour of a combinator-based graph reducer can often be approximated by a very simple model. An example of this can be found in Table IV, where the depth of the average tree was used to approximate the depth of nodes of a graph.

For most programs the graph grows in an initial phase to a certain size around which it fluctuates for the major part of the computation and which is then reduced to the result in the final phase. Usually the central phase accounts for more than 80 per cent of the reductions, during which the size fluctuations are less than 20 per cent. In the central phase less than 10 per cent of the nodes are shared (i.e. have a reference count greater than one). This leads to the conclusion that, as in SKIM II, the one-bit reference counting technique\textsuperscript{24} (with the count stored in the pointer) could have been used successfully in our SASL implementation. This implementation produces on the average one small cycle in a few hundred nodes. At the cost of some extra reduction steps the number of cycles can be reduced by orders of magnitude. About one node is claimed and another one released per reduction step. Most nodes only live a short time.

Table V shows that more than 94 per cent of the nodes in the graphs of the four medium-sized programs represent structure, not pure data values. If this is also true for really large programs, a representation in which much more structure is encoded implicitly could yield tremendous savings, both in time and in storage. Reduction systems based on super or serial combinators are steps in this direction.\textsuperscript{4,6} The optimization by the special constructor nodes could be extended to include nodes for other data structures, such as arrays and records. The disadvantage of such an approach is that the storage allocation and reclamation system will have to be able to cope with nodes of arbitrary size. The compensation for the extra effort to support nodes of arbitrary size must come from savings in space and/or time. However, the results in Table VI indicate that long lists are relatively rare. The introduction of cells of arbitrary size in the SASL implementation would therefore not be worth while. Instead, an additional type of node with, for instance, a maximum of four pointers could be considered. These results should be interpreted with care. The SASL implementation is fully lazy and evaluation is driven by the need to print. Therefore, unless a list is shared, most of the time only a small part of it will be present. In an implementation that supports eagerly-evaluated data structures the situation is very different.

ACKNOWLEDGEMENTS

Some of the ideas worked out in this paper arose in discussions with Betsy Pepels. Wim Vree and two anonymous referees made valuable comments on draft versions of the paper. We used the compiler developed by Riet Oolman. Her help and that of Henk Keller during the development of the instrumentation software were greatly appreciated. Lex Augusteijn and Theo van der Lee translated the paraffin program from KRC into SASL. This work is sponsored by the Dutch ministry of Science and Education, dienst Wetenschapsbeleid.
REFERENCES