ALSO PUBLISHED IN THE PROC. OF THE THIRD WORKSHOP ON THE PARALLEL IMPLEMENTATION OF FUNCTIONAL LANGUAGES (1991), TECHNICAL REPORT SERIES, CSTR 91-07, JUNE 1991, UNIVERSITY OF SOUTHAMPTON, UK.

EFFICIENT HEAP MANAGEMENT IN THE CONCRETE ABC MACHINE

John van Groningen, Eric Nöcker, Sjaak Smetsers

Faculty of Mathematics and Computer Science,
University of Nijmegen,
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
E-mail: clean@cs.kun.nl

June 1991

Abstract
This paper gives a description of the heap management system used in the implementation of Concurrent Clean. The compilation of Concurrent Clean to concrete target machine code is done with the aid of the abstract ABC machine. The ABC machine is a stack based graph reduction machine. The way graphs can be represented and manipulated on a real machine efficiently will be discussed. An important part of the heap management system is formed by the garbage collection algorithm: a combination of a copying and a sliding compaction garbage collector. It will be shown that a smart node representation together with the proposed garbage collector makes it is possible to run large functional programs on rather small machines.

1. Introduction

The current state of the affairs with respect to the implementations of functional languages is that, in contrast with the past, the time efficiency of functional programs has been improved significantly. Executing programs written in functional languages, such as LML (Augustsson & Johnsson (1989)), Hope (Burstall et al. (1980)) and Concurrent Clean (Nöcker et al. (1991)), shows that the current implementations of these languages are even able to compete with implementations of imperative languages such as C. However, a disadvantage of these languages is that, when executed, functional programs tend to consume large amounts of memory in a very unpredictable way. The main reason for this is that the lazy evaluation scheme of functional languages is, as far as the usage of space is concerned, far from optimal. Besides that, the memory management has to be controlled fully by the run-time system. A consequence is that most implementations of functional languages are only available on large machines. An exception to this is the Concurrent Clean system. Due to its elaborate and efficient memory management it is possible to execute large programs on rather small micro computers (such as a Macintosh Plus equipped with only 2.5M of RAM).
In this paper we will discuss the memory management part of the Concurrent Clean system. The description will be given with the aid of the abstract ABC machine which is used for the compilation of Concurrent Clean Brus et al. (1987), (Smetsers (1989), Eekelen et al. (1990)). The ABC machine (Koopman et al. (1990)) is a stack based graph reduction machine, similar to advanced G-machine like architectures (e.g. Johnsson (1987), Peyton Jones & Salkild (1989)). The actual target processor for implementing the ABC machine is the Motorola MC68020 processor. Due to the similarity of both the ABC machine with other abstract machines and the MC68020 processor with other register based processors, it will not be difficult to use the presented ideas in other implementations.
Note that this paper contains only a description of the heap management. A complete description of the compilation of Concurrent Clean to MC68020 code can be found in Smetsers (1991). It includes, for example, a discussion of the parameter passing mechanism on the ABC level and a register allocation algorithm based on basic block analyses.

Overview of the paper

In the rest of this introduction we give a very short overview of the ABC machine. Then we describe how the basic components of the ABC machine are mapped onto the MC68020 (Section 2). In Section 3 we present a very efficient implementation of a garbage collector. Finally, we discuss the effects of the heap representation and the garbage collection algorithm with the aid of a few benchmarks (section 4) .

The ABC Machine

Since a complete, formal description of the ABC machine goes far beyond the scope of this paper, we will restrict ourselves to a short introduction. In the sequel, specific parts of the machine will be highlighted further if necessary.
The ABC machine is a stack based graph reduction machine. Its main parts of interest are the three stacks (A, B and C stack) and the heap. The C stack is used for storing code addresses. The other two stacks are used for evaluating or building expressions, for passing arguments to functions and for returning results from functions. The A stack contains addresses of nodes in the heap, whereas the B stack contains values of basic types, such as integers or reals. Thus, basic values can be represented in two ways: as a node in the heap or as an item on the B stack. Graphs, which consist of collections of nodes, are stored in the heap. A node in the ABC machine, which represents a node of a Concurrent Clean graph, has a variable size.

The structure of nodes

Generally spoken, a node of a Concurrent Clean graph consists of a symbol with a certain number of arguments. Representing a node as a variable sized object causes problems with updating: the new node does not need to fit in the space of old one. This problem can be solved by introducing indirection nodes, but this will slow down the access to the contents of a node. In the ABC machine a node is split in a fixed and a variable sized part. The fixed size part contains a descriptor, a code pointer and a pointer to a variable sized part.

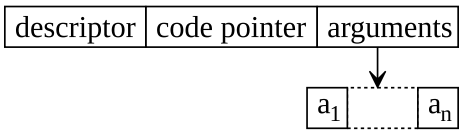


Fig 2.1 The node structure

The *descriptor* is a representation of a Clean symbol. Normally it is an index or pointer in a descriptor table. Descriptors are used for pattern matching, evaluating higher order functions and for fetching the arity of the node (for instance, during the garbage collection).

The code pointer refers to code with which the node can be evaluated to head normal form. This code is entered by a `jsr_eval` instruction. During reduction the code pointer can be changed. For example, after entering the node for evaluation a pointer to an error routine can be stored. If the node is ever entered again (indicating a non-terminating reduction) this code will be executed. If a node is updated with a head normal form value, the code pointer points to special code just containing a return statement:

```
  _hnf_code:              rtn
```

In the variable sized part the arguments of the node are stored. This means that the arguments have to be fetched via an extra indirection. On the other hand, updating a node is simple: update the fixed part, and allocate space for the arguments.

For nodes containing a basic value, e.g. an integer, the descriptor does not represent the Clean symbol (that would be the integer value itself). Instead, all integers share the same descriptor (e.g. INT). The integer value itself is stored in the argument reference part. For basic values that do not fit in the fixed part of a node (e.g. strings) a pointer to the value (for which space has to be allocated) is stored. Since basic nodes are always in normal form, they all contain the head normal form code pointer.

Higher order functions

In the ABC machine curried function applications are represented by *partial nodes*, i.e. nodes with a partially filled argument part. Such nodes are built as standard nodes, but contain special descriptors. Therefore, n+1 descriptors are defined for each Concurrent Clean symbol of arity n. In many respects, the ABC machine treats partial nodes in the same way as standard nodes. However, a partial node may be applied to another node. If that node needs exactly one more argument, all arguments are available: the function of the partial node can now be called. Otherwise, a new partial node, that is a copy of the original node with one extra argument, will be built.

2. Representing the ABC Machine

In this section we will describe how abstract ABC machine is mapped onto a concrete machine code. The target machine used in the description is based on the Motorola MC68020 processor.

The MC68020 processor contains besides a program counter and a status register two kinds of general purpose registers, to wit data and address registers, eight of each kind. The data registers, often indicated by d0-d7, are mainly used in arithmetical operations whilst the address registers (indicated by a0-a7) can be used to access data structures that are kept in memory. An important property of this processor is that the quality of the generated code strongly depends on how well registers are utilised.

The basic machine components

Mapping the components of the ABC machine (i.e. ABC stacks, graph store (heap)) onto the MC68020 does not cause many difficulties. Stacks can be implemented straightforwardly using some of the address registers. Implementing the heap takes some more doing.

The A, B and C stack

The A and B stack are allocated in one contiguous area of memory where they can grow in opposite direction. In this way a check on stack overflow of both A and B stack can be done with a few instructions (just compare the two stack pointers and check whether their difference is not negative). The pointers to the tops of the stack are held in registers: for the A stack register a3 is reserved, for the B stack register a4 (for convenience we will refer to those registers by *asp* and *bsp* from now on).

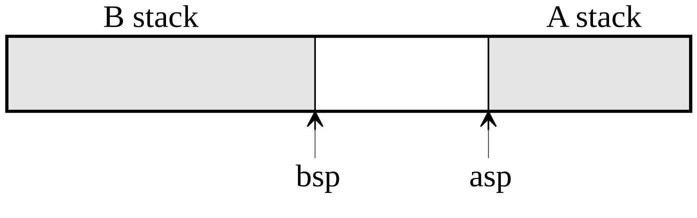


Fig 2.1: The layout of the combined A and B stack

For the C stack the system stack is taken (i.e. the stack used by the processor itself when performing a subroutine call). Therefore, the jump and return instructions of the ABC machine can be mapped directly on those of the MC68020 (of course, for the `jsr_eval` instructions other things have to be done, see further on). This implies that address register a7 (normally called *sp*) is reserved.

The descriptor table

We described how partial (curried) function applications are implemented on the ABC machine. In the implementation of the ABC machine, not only the actual arity of the curried application is stored in the descriptor but also a pointer to the code that should be executed when a partial application of the corresponding symbol is applied to an additional argument. In this way no tests are needed in order to decide whether there are sufficient arguments available.

This results in the following representation of symbol descriptors:

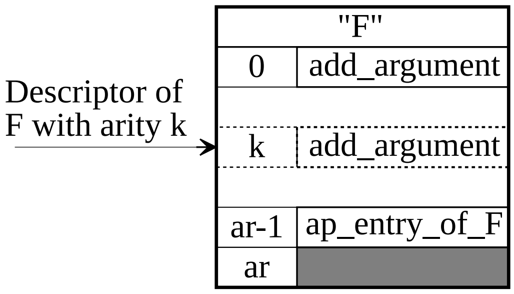


Fig 2.2: The lay-out of a descriptor in the descriptor table

`F` is a function with arity `ar`. The descriptor contains a string representation of the name and `ar`+1 entries. Now, the pointer stored in the descriptor field of the node is just a reference to the entry that corresponds to the arity with which F is actually applied. Besides this actual arity that is used by the garbage collector, each entry contains the code that should be reduced when a partial application of F is applied to an additional argument. It should be clear that this code is just the `add_argument` code (i.e.the code that copies the node and adds an argument to it) unless the curried application has already `ar`-1 arguments. In that case the extra argument provides that this particular application becomes complete. So the apply entry of `F` can be called.

The descriptors of all symbols defined in a Clean program are stored in a so-called *descriptor table.* As a consequence, each application of a symbol F with actual arity k can be represented by an offset in this table that corresponds to the k-th entry of the descriptor of F.

With the aid of the previous representation the translation of the apply code will result in the following ABC instructions (we assume that register `a1` refers to the node containing the partial application and register *st* refers to the beginning of the symbol table):

```
move 2(a1), a2 ; get the offset of the descriptor entry
add.l  st,    a2; add this offset to the beginning of the descriptor table
move.l 2(a2), a2; retrieve the reduction code
jsr    (a2)       ; call the reduction code
```

The heap

For the heap a contiguous area of memory is reserved. The pointer to the free area is stored in register a6 (called *hp*), whereas the number of free heap cells (1 heap cell = 1 long word = 4 bytes) is stored in register d7 (*fh* from now on). With this representation the allocation of memory becomes cheap. Also, the filling of newly created nodes in the heap can be done efficiently (for an example see (Smetsers et al. (1991)).

Memory is recycled by a process called *garbage collection*. In our implementation a combination of two kinds of garbage collectors is used: a *copying* and a *sliding compaction* garbage collector. Both collectors and there combination are described in section 3.

Representation of nodes

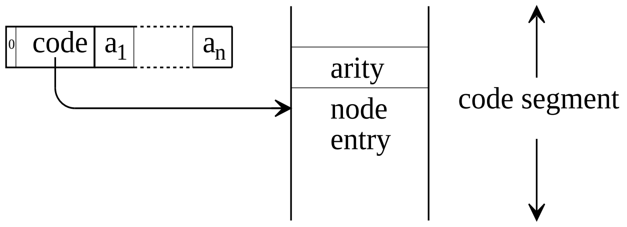
As described earlier, a node in the ABC machine consists of a fixed and a variable sized part. The fixed size part consists of a pointer to a descriptor, a code pointer and a pointer to the variable sized part. A drawback of the ABC node structure is that the size of the nodes is relatively large: the fixed part would consist of 3 long words (12 bytes), one long word for each pointer. It is important that nodes are as small as possible: because less memory is consumed the garbage collector will be called less often and also the filling and copying of nodes can be done faster.

The following observations make a more efficient representation of nodes possible:

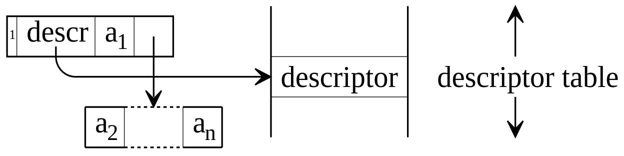
- If a node is in head normal form, its code field points to the head normal form code so in fact only the pointer to the descriptor is of interest. On the other hand, if a node contains an (unevaluated) expression the descriptor is not used (from now on we will call these nodes *closures*). This allows us to combine the descriptor and code field into one reducing the size of the fixed part by one third. However, one little problem has to be solved: the arity of a node is needed by the garbage collection. This arity, stored in the descriptor table, is not accessible when the descriptor is no longer available. The problem is solved by storing the arity not only in the descriptor table but also just before the node entry such that it can be accessed via the code pointer.
- Many nodes in head normal form have less than 3 arguments (for instance, the list constructors Cons and Nil). For this reason we have decided to create a variable sized part only if there are more than 2 arguments. Note that such nodes are also large enough to hold integers and reals.
- Only closures are overwritten (i.e. once a node is in head normal it remains unchanged forever). Furthermore, a closure is never overwritten with another closure. A consequence is that for the arguments of closures an additional argument part is not necessary: the arguments can be stored in the fixed part itself, provided that the whole node is large enough to contain the fixed part of a node in head normal form. The means that closures always have a minimal node size of 3 long words which implies that for nullary and unary functions resp. two long words and one long word are spilled.

The new node structure is illustrated in the next two pictures. A disadvantage is that each node has to be supplied with a tag: the highest bit of the first word of the code/descr field (note that this field consists of two words) indicates whether it contains a descriptor or a code address. If this bit is set, the second word is an index in the descriptor table. Otherwise, the code/descr field contains a code pointer that is used to reduce the node to root normal form.

Ordinary representation of nodes:



Closures with arity ≥ 2



Head normal forms with arity ≥ 3

Fig 2.3: The structure of nodes

Examples of alternative representations:

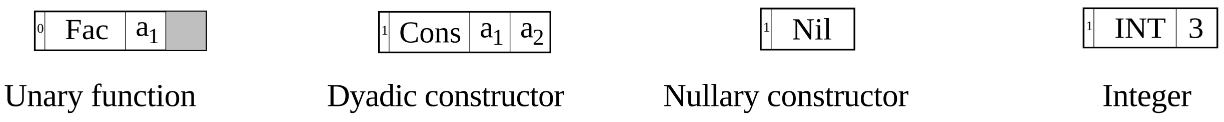


Fig 2.4: Examples of non-standard nodes

3 Garbage Collection

Garbage collection is based on a combination of two well-known schemes: a copying garbage collection and a mark-scan garbage collection mechanism. We will not treat the basic ideas of these algorithms, as this is out of the scope of this paper (good descriptions can be found in Sansom (1991) for example). First, we will outline why for such a combination is chosen. However, the technique described depends on the efficiency of the implementation. So, in the next two sections some implementation details are discussed, for which we assume knowledge about the precise aspects of these garbage collection algorithms. A detailed description of the implementation of the garbage collectors can be found in Groningen (1991).

For a copying collector the heap is divided into two equally sized areas (*semispaces*). Memory is allocated in one semispace. When this area is filled up, the garbage collector copies all accessible objects to the other semispace, leaving all the garbage behind. This scheme has some well-known positive properties:

- only the non garbage objects are visited. So, if not much heap space is used, the garbage collection will be very fast.
- it is very easy to implement, that is, it is relatively easy to obtain an efficient implementation. For example, it can be written directly in assembly.

Unfortunately, it has one major disadvantage: only half of the available memory can be used. This disadvantage becomes very clear if continuously almost half of the heap space is allocated: though still enough heap space is available, very much time can be spent on garbage collection. In such a case it might be better to use a garbage collector that is slower, but that can use the whole heap. It appears to be possible to switch to a mark-scan copying garbage collector at a certain point of heap usage.
The mark-scan variant we will consider is based on Jonker's sliding compaction algorithm (Jonker (1979)). It works in two phases:

- in the first phase (marking) all accessible heap objects are marked. This is done by traversing all graphs, reachable from some collection of root nodes.
- in the second phase (scanning) all marked objects are scanned. Two scans are required. In the first scan all forwarding pointers (i.e. pointing to an object with a higher address) are updated with the addresses of the new locations of the objects. All pointers pointing backward to a particular object are linked in a list starting from that object. In the second scan all objects are moved to their new locations. Also the backward pointers are updated with their new values.

This algorithm has two major disadvantages:

- the whole heap has to be scanned twice. So, the collection time is proportional to the size of the heap. Even if the number of accessible objects is small, the collection can take some time.
- for each object rather much work has to be done. The administration of the pointers is complex.

In general, this garbage collector is slower than the copying one. But if a certain amount of the heap is filled, it will be faster. In Sansom (1991), where a similar idea is presented, it is shown that the turning point r is at:

$$r = \frac{k/2 - 1}{k - 1}$$

where k is the ratio between the speeds of the copying of objects of both collectors. For example, if the mark-scan collector is 3 times as slow, the turning point is at 25% heap use. In our implementation we have measured this particular value for k.

Implementing a Copying Garbage Collector

As mentioned before, it is quite straightforward to implement a copying garbage collector. This holds even if a wide variety of nodes is possible. However, in the ABC machine there is one point to take account of. It is important that a garbage collection always succeeds (otherwise, the heap might be in such a state that it is impossible to switch to the other collector). In the ABC machine it is possible that argument vectors are shared between nodes (because of the `fill_a` instruction). In a simple implementation of the garbage collector this sharing is lost. In that case it is possible that not enough heap is available for making the copy. This can be solved by using forwarding pointers to the new argument vectors.

There are not many possibilities to optimise this garbage collector. Yet, there is one optimisation that has to be mentioned. In principle, all nodes will be visited twice: once in the old semispace, for making the copy and storing a forwarding reference, and once in the new semispace for copying the arguments. This is not necessary for nodes without arguments (e.g. Integers or Nil). If such nodes are stored at the end of the new semispace, they don't need to be scanned the second time.

Implementing a Sliding Compaction Garbage Collector

The mark-scan algorithm is rather inefficient. The main causes are the inefficient marking, and the two scans where complex pointer manipulation has to be done. We will give some improvements, such that an acceptable performance is obtained. In first instance, we will assume one type of nodes: each node consists of a descriptor field (which is large enough to hold a pointer, and which is always even), followed by some arguments. Thereafter, we will discuss the particular problems caused by other kinds of nodes.

The first improvement is obtained by rearranging the phases. It appears to be possible to combine the marking phase and the creation of the lists of backward pointers (of the first scan). In the second phase only one scan is needed, in which the nodes are copied, and all the pointers are updated.

Marking the graph requires a traversal of the graph. This can be done by using a pointer reversal algorithm. The optimal way of traversing is depth-first, where the arguments are visited from right to left. Then only the first argument needs to be marked in order to detect that all the arguments have been handled. After marking all arguments, the address of the node itself can be found without using the arity. Pointers can be marked easily by using their least significant bit (addresses are always even).The backward list is constructed during the upward traversal.
The marking itself can be done by using a bit vector: each long word in the heap corresponds with one bit. So, the nodes itself contain no mark field. For this bit vector 1/33 of the heap has to be

reserved. In the scanning phase it is rather easy to find the marked nodes corresponding to the bits that are set. The MC68020 has instructions with which these bit vectors can be manipulated very efficiently.

At the end of the first phase all accessible nodes are marked, and the descriptor fields of those nodes contain a list of all backward pointers to that node (ending with the value of the descriptor of the node, which can be distinguished because it is not marked). Now, the first node can safely be moved to its new position, since there are no forwarding pointers to this node. All backward pointers to this node are updated with the new position of the node, and the descriptor is stored in the new node. Since there might be a forwarding pointer to the next node, it cannot be copied immediately. This is solved by adding all forwarding pointers of the first node to the appropriate lists of backward pointers. By repeating this, all nodes can be moved to their new positions.

For nodes that are split in two parts this algorithm has to be changed slightly. This can be done by considering argument vectors as nodes also. However, since the layout of ordinary nodes and argument vectors differs, the garbage collector has to be able to distinguish between them. For the marking this is no problem, since argument vectors are always visited via the fixed sized part of a node. During the generation of the backward lists and in the second phase the garbage collector cannot see the difference anymore. Another problem is that for an argument vector there is no data field in which a list of backward pointers can be stored. This implies that the first argument of an argument vector can appear in two different lists of pointers: in the list of pointers refering to this vector and in the list of pointers belonging to the node to which the first argument itself refers. This can be solved by appending the second list to the first one, and marking all the pointers in the first list. Since nodes are aligned on long word boundaries, addresses will always be a multiple of 4, so that the second least significant bit can be used for this. Although the algorithm becomes more complex, it is hardly slower since argument vectors are rarely shared.

Garbage collecting with code pointers

A much-discussed technique for implementing garbage collection, that is supposed to be efficient, uses code pointers. Besides the ordinary code for evaluating closures, the compiler generates additional code that is called by the garbage collector when a closure is encountered during the garbage collection. Among others, this technique has been used in Peyton Jones (1991) and Sansom (1991). It will be clear that the main advantage of this method is that is requires no run-time tests for arities. Loops needed to handle all the arguments of a certain node can be avoided by unfolding them in the additional code. Apart from the doubts that we have whether the gain in speed is significant, there is one major disadvantage of this method: it may lead to a (sometimes unacceptable) increase of code (note that the increase of code is proportial to the number of functions appearing in a program). Things are getting even worse if the proposed combined garbage collector is used. In Sansom (1991) it is shown that in this case several entry points are necessary each corresponding to one of the garbage collection phases. An advantage of an 'interpreting collector' is that it can be optimised as much as possible using the the full power of the target machine. Furthermore, adjusting and experimenting is certainly much easier with the interpreting collector than with the other collector. All in all, we believe that the advantages of using code pointers are outweighed by the advantages of the interpreting collector.

4. Discussion

In this section we analyse the proposed optimisations briefly with the aid of two example programs. The first program is called fastfourier, an algorithm that computes the fast fourier transformation of a list of 8K complex numbers. The second program, called reverse, reverses a list of 3K elements 3K times.

The first table (Table 3.1) shows the results of comparing an unoptimised node representation with the representation as presented in section 2. The times in the table are execution times given in seconds. In both programs only the copying garbage collector has been used.

	old	new
fastfourier	23	17
reverse	63	57

Table 3.1

The second table (Table 3.2) compares copying with sliding compaction garbage collection. The example programs have been executed with a number of heap sizes. The times given are times for the garbage collections alone, again given in seconds. The actual execution times are not included. The table also gives the results when using the combined garbage collection algorithm.

	copying gc	sliding compaction gc	combined gc
fastfourier			
1M	OutOfHeap	11	11
1.5M	8.3	5.4	5.2
2M	4.0	3.7	3.4
3M	2.1	2.1	2.1
reverse			
0.25	59	57	57
1M	9.0	12	8.9
2M	4.5	6.2	4.2

Table 3.2

When decreasing the heap size the compaction garbage collection clearly defeats the copying collector. It should be noticed that even when using a large heap the copying collector is not much faster than the other one. This can be explained as follows: the copying garbage collector uses effectively only one half of the heap, so it has to be called twice as often as the compaction garbage collector. The amount of memory that is non garbage is for both cases less than half of the memory (otherwise the algorithm would fail to terminate correctly). Due to its highly optimised scanning of the bit vector the compaction garbage collector is able to find all these non-garbage nodes very fast. This implies that the overhead that is involved in scanning the whole bit vector (of which the size is proportional to the heap size) and skipping all the garbage nodes is small. The combination of both algorithms confirms our expectations: in general, it is at least as fast as the best of the two algorithms.

We truly recommend to use the combined garbage collection algorithm: it improves the performance of the program in all cases. Of course, having two different algorithms for garbage collection causes an increase of the size of the run time system. However, the overhead due to this is negligible when compared with the code generated for an average functional program. For, the code of each garbage collector occupies only about 1.2K of memory. The alternative node structure described in section 2 is worth considering. Measurements have shown that the gain in execution speed varies from 10 to 50 percent. It should be pointed out that implementing these representations may take some doing: not only the instructions working on nodes have to be adjusted but also the garbage collectors become more complicated.

References

Augustsson L., Johnsson T. (1989), 'The Chalmers Lazy-ML Compiler', The Computer Journal, Vol. 32, No. 2 1989.

Barendregt, H.P., Eekelen, M.C.J.D. van, Glauert, J.R.W., Kennaway, J.R., Plasmeijer, M.J., Sleep, M.R. (1987), 'Term Graph Reduction', Proceedings of Parallel Architectures and Languages Europe (PARLE), part II, Eindhoven, The Netherlands, LNCS Vol. 259, pp. 141-158, June 1987.

Brus T., Eekelen M.C.J.D. van, Leer M. van, Plasmeijer M.J. (1987), 'Clean - a Language for Functional Graph Rewriting', Proc. of the third International Conference on Functional Programming Languages and Computer Architecture (FPCA '87), Portland, Oregon, USA, Springer Lecture Notes on Computer Science 274, pp. 346-384.

Burstall, R.M., MacQueen, D.B., and Sanella, D.T. (1980), 'Hope: An Experimental Applicative Language', Proceedings of the 1980 LISP Conference, 136 - 143.

Cheney C.J. (1970), 'A nonrecursive list compacting algorithm', CACM 13, November 1970, pp. 677-678.

Eekelen, M.C.J.D. van, Nöcker E.G.J.M.H., Plasmeijer M.J., Smetsers J.E.W. (1990), 'Concurrent Clean, version 0.6', Technical Report 90-21, University of Nijmegen, December 1990.

Groningen J.H.G. van. (1991), 'Implementing the ABC-machine on M680x0 based architectures'. Technical Report, Department of Computer Science, University of Nijmegen, to appear in 1991.

Johnsson Th. (1987), 'Compiling Lazy Functional Programming languages'. Dissertation at Chalmers University, Göteborg, Sweden. ISBN 91-7032-280-5.

Jonker H.B.M. (1979), 'A fast garbage compaction algorithm', Info. Proc. Lett. 9, July 1979, pp. 26-30.

Koopman P.W.M., Eekelen M.C.J.D. van, Nöcker E.G.J.M.H., Smetsers J.E.W., Plasmeijer M.J. (1990). 'The ABC-machine: A Sequential Stack-based Abstract Machine For Graph Rewriting'. Technical Report no. 90-22, December 1990, University of Nijmegen.

Nöcker E.G.J.M.H., Smetsers J.E.W., Eekelen, M.C.J.D. van, Plasmeijer (1991). 'Concurrent Clean', Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE '91), Eindhoven, The Netherlands, Lecture Notes on Computer Science, Springer Verlag, to appear in June 1991.

Peyton Jones S.L, Salkild J. (1989). 'The Spineless Tagless G-machine'. Proceedings of the Conference on Functional Programming Languages and Computer Architectures, Addison Wesley, pp 184 - 201.

Peyton Jones S.L (1991), 'The spineless tagless G-machine: a second attempt', Proc. of Third International Workshop on Implementation of Functional Languages on Parallel Architectures, Technical Report Series CSTR91-07, June 1991, Department of Electronics and Computer Science, University of Southampton, UK, pp. 147-192.

Sansom P.M. (1991), 'Dual-Mode Garbage Collection', Proc. of Third International Workshop on Implementation of Functional Languages on Parallel Architectures, Technical Report Series CSTR91-07, June 1991, Department of Electronics and Computer Science, University of Southampton, UK, pp. 283-310

Smetsers J.E.W., (1989). 'Compiling Clean to Abstract ABC-Machine Code', University of Nijmegen, Technical Report 89-20, October 1989.

Smetsers J.E.W., Nöcker E.G.J.M.H., Groningen J.H.G. van., Plasmeijer M.J. (1991), 'Generating Efficient Code for Lazy Functional Languages', Proc. of the International Conference on Functional Programming Languages and Computer Architecture (FPCA '91), Boston, USA, Springer Lecture Notes on Computer Science, Springer Verlag, to appear in 1991.