Implementing the ABC-machine

on M680x0 based architectures

John van Groningen

November 1990

Master's thesis 168

University of Nijmegen

Faculty of Mathematics and Informatics
Department of Informatics

Preface

This Master's thesis has been written as the final stage of my studies in Computer Science at the University of Nijmegen. The research of which this thesis is the result has been carried out from February 1990 until November 1990 under guidance of Drs. J.E.W. Smetsers and Drs. E.G.J.M.H. Nöcker, whom I would like to thank for their help.

John van Groningen
Nijmegen, November 1990

Abstract.

To compile the functional language Clean, first code is generated for an abstract machine, the ABC machine. This ABC code is then translated to the concrete machine by a code generator. Such a code generator, which generates code for the MC68020 processor, has been designed and implemented and is described here.

To generate code first the ABC instructions are divided into basic blocks. Then is determined which values are stored in registers at the start of such a basic block. Parameters and results of functions are passed in registers. Then a graph is constructed for the basic block, which represents the computations performed by this basic block. After that the code generator determines which values are stored in registers at the end of the basic block.

Then the order in which this graph will be evaluated is determined using an adapted labeling algorithm. This adapted labeling algorithm determines a better evaluation order than the original labeling algorithm if some values are stored in registers at the beginning and/or end of the basic block, and can handle common subexpressions.

Then intermediate code is generated from the graph. This intermediate code is very close to MC68020 machine code, but in this intermediate code an unlimited number of address and data registers may be used.

While constructing the graph and generating intermediate code several optimizations are performed. Then the local register allocator changes the intermediate code of a basic block so that no more than 8 address registers and 8 data registers (the MC68020 has 8 data registers and address registers) are used.

Then the intermediate code is optimized by using the postincrement and predecrement addressing modes of the MC68020 and optimizing jumps.

And then from this intermediate code object code is generated for the linker. While generating this object code many very simple MC68020 specific peephole optimizations are performed.

Compared to the previous (simpler) code generator computations on strict arguments and basic values (integers, reals, etc.) are done a lot faster (about 1.3 - 2.9 times as fast for some benchmarks). But computations on non strict arguments are executed only a bit faster (about 1.1 - 1.5 times as fast). Curried function applications are also executed a lot faster (about 1.8 - 2.0 times as fast), but not so much due to better code generation techniques.

Contents.

1. Introduction. 1
2. Description of languages and machines. 3
2.1. Clean. 3
2.2. The ABC machine. 4
2.2.1. ABC instructions for graph manipulation. 6
2.2.2. ABC instructions for information retrieval from nodes. 6
2.2.3. ABC instructions for pattern matching. 7
2.2.4. ABC instructions to manipulate the A- and B-stack. 7
2.2.5. ABC instructions to push constants on the B-stack. 8
2.2.6. ABC instructions to change the flow of control. 8
2.2.7. ABC instructions to generate output. 8
2.2.8. ABC instructions to implement delta-rules. 9
2.3. The MC68020 microprocessor. 9
2.3.1. The MC68020 registers. 9
2.3.2. The MC68020 data types. 10
2.3.3. The MC68020 addressing modes. 10
2.3.4. The MC68020 instruction set. 11
2.3.5. Example of MC68020 code of a Clean function. 16
2.3.6. The MC68020 cache. 17
2.3.7. MC68020 instruction execution timing. 17
3. Representing the ABC machine. 20
3.1. The stacks. 20
3.2 The heap. 20
3.3. Representing nodes. 20
3.4. Representing strings. 23
3.5. Representation on the MC68020. 24
4. Run time system. 27
4.1. Garbage collection. 27
4.2. Remainder of the run time system. 27
5. Possible code optimizations. 29
5.1. General code optimizations. 29
5.1.1. Optimizing the creation of nodes. 29
5.1.2. Jump optimization. 31
5.1.3. Strength reduction. 31
5.1.4. Constant folding. 32
5.1.5. Other algebraic optimizations. 32
5.1.6. Common subexpression elimination. 32
5.1.7. Removal of unused code. 33
5.1.8. In-line code substitution for small functions. 33
5.1.9. Removing pointers from the A-stack. 33
5.2. Possible code optimizations for register machines. 33
5.2.1. Better use of registers. 34
5.2.2. Passing parameters and results of functions in registers. 34
5.2.3. Eliminating unnecessary copies. 35
5.2.4. Optimizing booleans. 35
5.2.5. Changing the evaluation order. 36
5.2.6. Optimizing jsr_eval instructions. 37
5.3. MC68020 specific code optimizations. 37
6. Generating code. 39
6.1. Constructing a dag for every basic block. 39
6.1.1. Conditions for changing the evaluation order. 40
6.1.2. Problems on a stack machine when changing the evaluation order. 40
6.1.3. Simulating the A- and B-stack. 41
6.1.4. Local variables. 42
6.1.5. Dags (directed acyclic graphs). 42
6.1.6. Remaining ABC instructions without side effects. 43

6.1.7. ABC instructions with side effects. 45
6.1.8. Division into basic blocks. 46
6.1.9. Constructing the dag. 47
6.2. Determining the evaluation order of the dag. 47
6.2.1. The evaluation order for trees. 48
6.2.2. The labeling algorithm. 48
6.2.3. The dynamic programming algorithm. 49
6.2.4. Using these algorithms to generate code for the MC68020. 51
6.2.5. The evaluation order for dags with common subexpressions. 52
6.2.6. The evaluation order for dags with variables in registers. 54
6.2.7. Calculation of the evaluation order of the arguments for a machine with one type of register. 55
6.2.8. Calculation of the evaluation order of the arguments for the MC68020. 58
6.2.9. The evaluation order for dags with common subexpressions on the MC68020. 59
6.3. The dag representation of the ABC instructions. 60
6.3.1. The sort of dag representation. 60
6.3.2. The dag representation for arguments. 61
6.3.3. The dag representation for operations. 61
6.3.4. The dag representation for floating point arguments. 62
6.3.5. The dag representation for floating point operations. 63
6.3.6. The dag representation for storing values on the stack and in registers. 63
6.3.7. The dag representation for push_args and repl_args. 63
6.3.8. The dag representation for create and del_args. 65
6.3.9. The dag representation for fill instructions and set_entry. 66
6.3.10. The dag representation for add_args. 66
6.4. Generating intermediate code from the dag. 67
6.4.1. Calculating reference counts. 67
6.4.2. Choosing between address registers and data registers. 68
6.4.3. Register allocation during code generation from the graph. 69
6.4.4. The intermediate code. 69
6.4.5. Generating intermediate code from the dag. 69
6.4.6. Generating code for arithmetic dyadic operation nodes. 70
6.4.7. Generating code for STORE and FSTORE nodes. 71
6.4.8. Generating code for STORE_R nodes. 71
6.4.9. Generating code for MOVEM and MOVEMI nodes. 72
6.4.10. Generating code for FILL nodes. 73
6.4.11. Generating code for CREATE nodes. 73
6.4.12. Optimizing the creation of nodes. 74
6.4.13. Using the condition codes of the MC68020. 74
6.4.14. Preventing unnecessary stores. 75
6.4.15. Optimizing the use of small constants. 75
6.5. Global register allocation. 76
6.5.1. Directives describing the parameters and results for the code generator. 76
6.5.2. Function calling convention. 77
6.5.3. Conditions for global register allocation. 78
6.5.4. Consequences of the function calling convention for global register allocation. 79
6.5.5. Straightforward global register allocation. 80
6.6. Local register allocation. 82
6.6.1. Local register allocation strategy. 82
6.6.2. Local register allocation algorithm. 83
6.6.3. Preserving condition codes during local register allocation. 86
6.7. Optimizing stack accesses. 87
6.8. Optimizing jumps. 89
6.9. Calling the garbage collector. 90
6.10. Generating MC68020 code from the intermediate code. 91

7. Evaluation. **92**
7.1. Implemented optimizations. 92
7.2. Possible improvements. 92
7.3. Comparing this code generator with the previous code generator. 93
APPENDIX A: Examples. **96**
A.1. Fac. 96
A.1.1. Fac in Clean. 96
A.1.2. ABC code of fac. 96
A.1.3. Basic blocks of the ABC code of fac. 97
A.1.4. Dag representation and global register allocation of fac. 98
A.1.5. Dag representation of fac after computing the increases and uses of the number of registers and global register allocation. 100
A.1.6. Intermediate code of fac generated from the dag. 103
A.1.7. Intermediate code of fac after stack access optimization. 104
A.1.8. Intermediate code of fac after stack access optimization and jump optimization. 105
A.1.9. MC68020 code of fac. 106
A.2. Append. 107
A.2.1. Append in Clean. 107
A.2.2. ABC code of append. 107
A.2.3. Basic blocks of the ABC code of append. 108
A.2.4. Intermediate code of append after stack access optimization and jump optimization. 109
A.2.5. MC68020 code of append. 111
A.3. Inc. 113
A.3.1. Inc in Clean. 113
A.3.2. Basic blocks of the ABC code of inc. 113
A.3.3. Intermediate code of inc after stack access optimization and jump optimization. 115
A.3.4. MC68020 code of inc. 118
APPENDIX B: MC68020 cache case execution times. **122**
APPENDIX C: The nodes in the graph. **124**
APPENDIX D: The instructions of the intermediate code. **126**
APPENDIX E: Object file format. **127**
REFERENCES. **130**

1. Introduction.

The code generator which is described in this paper is part of the implementation of the graph rewriting language Clean [Brus et al. 1987]. To compile Clean, first the language Clean is compiled to ABC code [Koopman et al. 1990] by the Clean compiler [Smetsers 1989]. Then the ABC code is compiled to the machine code of the target machine. So for every target machine a code generator has to be written, but because the ABC code is machine independent, the same Clean compiler can be used for all target machines. The ABC code can also be executed by an interpreter.

The languages Clean and ABC have also been extended to be able to make efficient implementations for machines with more processors by concurrent execution, these languages are called Concurrent Clean [Nöcker et al. 1991, Smetsers et al. 1991] and PABC [Nöcker 1989].

When I started to design this code generator, a Clean compiler [Smetsers 1989] had been implemented which could be executed on the Macintosh and the Sun, and a simple code generator for the Sun [Weijers 1990] and an interpreter for the Macintosh [Nöcker 1989] had been implemented. This Clean compiler could also compile Concurrent Clean and the interpreter could simulate concurrent execution of PABC code.

The ABC code generator described here generates code for the Motorola MC68020 processor [Motorola 1985] and the floating point coprocessor MC68881 for the Macintosh II and has been implemented. But with some small changes the code generated by this code generator can be used for any machine with an MC68020 or MC68030 processor. And because only a few MC68020 specific instructions are used, the code generator can easily be changed to generate code for the MC68000 processor.

Briefly, this code generator generates code in the following way. First the ABC instructions are divided into *basic blocks*. Basically, a basic block is a sequence of ABC instructions of which only the last instruction may be an *instruction with side effects*. This last instruction of a basic block usually is a jump, branch, return or subroutine call instruction.

Then the code generator determines which values are stored in registers at the start of the basic block. Parameters and results of functions are passed in registers if enough registers are available.

And then a graph is constructed for such a basic block, which represents the computations performed by this basic block.

Because the MC68020 processor has two types of registers, i.e. data registers and address registers, a counter is maintained for every node during construction of the graph. These counters are used to determine whether to use a data register or an address register if a register is used.

Then the code generator determines which values are stored in registers at the end of the basic block. Because the graph is constructed so that no values are stored in registers, some parts of the graph have to be changed so that values are stored in registers at the end of the basic block.

Then the order in which this graph will be evaluated is determined using an adapted *labeling algorithm*. The three main differences between the original labeling algorithm and my adapted labeling algorithm are:

1. The original labeling algorithm assumes the graph is a tree. But my adapted labeling algorithm assumes the graph is a directed acyclic graph. In this way nodes may be shared, and common subexpressions can be represented in the graph.

2. The original labeling algorithm calculates for every node in the graph the number of registers necessary to evaluate the subgraph with as root this node. My adapted labeling algorithm calculates this number of registers as well, but also calculates by how many registers the number of used registers increases (or decreases) when the subgraph with as root this node is evaluated. In this way usually a better order to evaluate a graph is determined if some values are stored in registers at the beginning and/or end of the basic block. And if the graph contains shared nodes a reasonable evaluation order can be determined.

3. Because the MC68020 processor has two types of registers, for every node for both the address and the data registers the number of used registers and increase in the number of used registers are calculated. So four number of registers are calculated for every node.

Then an evaluation order is determined using these four numbers of registers in the nodes and an *intermediate code* is generated. This intermediate code is very close to MC68020 machine code. But in this intermediate code an unlimited number of address and data registers may be used.

While constructing the graph and generating code from the graph the following optimizations are performed: the creation of nodes by create instructions is optimized, use of booleans is optimized by using condition codes instead of booleans and many unnecessary copies and stack manipulations are eliminated.

Then the *local register allocator* changes the intermediate code of a basic block so that no more than 8 address registers and data registers (the MC68020 has 8 data registers and address registers) are used. It does this by changing the register numbers of the registers used by the intermediate instructions and by inserting instructions to load and store values in registers from/into memory.

Then accesses to the stacks are optimized by using the postincrement and predecrement addressing modes of the MC68020 by changing the intermediate code. After that jumps are optimized by replacing a branch instruction followed by a jump instruction by one branch instruction if possible.

And then from this intermediate code object code is generated for the linker. While generating this object code many very simple MC68020 specific peephole optimizations are performed.

Finally the linker produces an executable file for the Macintosh II from this object file, the object files of the other modules of this program (if any) and the library object files.

I will now briefly describe what is discussed in each chapter. In chapter 2 the languages and machines which you have to know to understand the rest of this paper are briefly described. These languages and machines are Clean, the ABC machine and the Motorola MC68020 processor. Also an example is given of a small Clean program, the ABC code generated by the Clean compiler from this program and the MC68020 code which is produced by my code generator from this ABC code. For the MC68020 also some execution times are given and some conclusions concerning what instructions to generate are drawn from these execution times.

In chapter 3 is described how the data structures (heaps, stack etc.) used by the ABC machine are represented on the MC68020. It describes how nodes are represented in the heap, and why this representation was chosen. Also the representation of strings is discussed.

Then, in chapter 4 the run time system is described. It explains how the garbage collector has been implemented, and what other things have been implemented in the run time system.

In chapter 5 possible code optimizations are described. General optimizations of ABC code, optimizations for register machines and MC68020 specific optimizations are discussed.

Then, in chapter 6 is described how the code generator generates code. First dividing the ABC instructions into basic blocks and determining the evaluation order by the adapted labeling algorithm is discussed. Then the graph representation is explained and how code is generated from this graph. Finally global register allocation, local register allocation, optimizations on the intermediate code, calling the garbage collector and generating MC68020 code from the intermediate code are discussed.

And finally in chapter 7 the code generator is evaluated and some possible improvements are described.

There are appendices with examples, instruction execution times, nodes in the graph, instructions of the intermediate code and the object file format.

2. Description of languages and machines.

In this chapter the languages and machines which you have to know to understand the rest of this paper are briefly described. These languages and machines are Clean, the ABC machine and the Motorola MC68020 processor. Also an example is given of a small Clean program, the ABC code generated by the Clean compiler from this program and the MC68020 code which is produced by my code generator from this ABC code. For the MC68020 also some execution times are given and some conclusions concerning what instructions to generate are drawn from these execution times.

2.1. Clean.

In this section I will briefly describe Clean. For this description of Clean I have used [Plasmeijer et al. 1989].

Clean [Brus et al. 1987] is an experimental language based on Functional Graph Rewriting Systems. Clean programs are purely functional. Clean stands for: Clean Lean. Lean [Barendregt et al. 1987] is another experimental language based on general Graph Rewriting Systems and stands for the Language of East Anglia and Nijmegen.

Clean is used in two ways. First of all it is intended as an intermediate language between arbitrary (eager and lazy) functional languages and arbitrary sequential machine architectures. In practise it is used as an intermediate language between Miranda and sequential architectures like VAX and Motorola. Secondly, Clean can also be seen as a simple functional programming language in which computations in terms of graph rewriting can be expressed.

A Clean program basically consists of a number of graph rewriting rules and a default initial data graph which can be rewritten to normal form according to these rules. The output of a Clean program is in principle a depth-first representation of the normal form to which the initial data graph is reduced. Clean supports run-time (pattern) matching. The notation used in Clean is the functional style. Variables begin with a lower-case character, constants may not begin with a lower-case character.

An example Clean program which computes the factorial of 20: (see appendix A for more examples)

```
MODULE Fac;

FROM deltaI IMPORT --I,*I;

RULE
      Fac 0       ->    1                  |
      Fac n       ->    *I n (Fac (--I n))      ;

      Start       ->    Fac 20            ;
```

(--I and *I are predefined rewrite rules on integers, which respectively subtracts one from an integer and multiplies two integers)

Clean is a typed language. The type system is based on the Milner-Mycroft type scheme [Milner 1978, Mycroft 1984]. Types can by specified explicitly for rewrite rules. If a rule is not typed, the type can often be derived by the Clean compiler. Basically, all types in Clean are algebraic types, which may be polymorphic. Type synonyms and abstract data types can also be defined. The predefined basic types are: INT (integer numbers), REAL (floating point numbers), CHAR (characters), BOOL (booleans), STRING (strings) and FILE (files). Denotations for lists and tuples are also predefined.

In Clean all symbols have a fixed arity, but curried applications of functions can be used. Curried applications are implemented using the predefined generic delta-rule AP. AP transforms a curried application of a function F with arity n into an uncurried application, if the curried F is applied to all n arguments. For AP the following rewrite rule is implicitly defined for every function F with arity n:

```
AP ( ... (AP (AP F a1) a2) ... ) an -> F a1 a2 ... an
```

These AP's are automatically introduced by the Clean compiler for every curried function application.

Clean performs graph rewriting, but graph rewriting is not very efficient. Therefore the Clean compiler tries to do as little graph rewriting as possible by not always building a node representation. It does this by storing evaluated integers, booleans, characters and reals on a stack and pass them to functions on the stack instead of in a node in the heap as much as possible, and by evaluating nodes as soon as possible. In this way the program is not only executed faster, but also uses less memory.

In order to perform these optimizations to obtain efficient code it is important to know whether or not a function is *strict* in its arguments. A function is strict in a certain argument if the evaluation of that argument is needed in the computation of the function result. Consequently, a strict argument can be evaluated in advance since its evaluation can not change the termination properties of the program. Strictness is, in general, an undecidable property. However, a good strictness analyser can find strict arguments in many cases. The Clean compiler can find strict arguments in many cases. [Nöcker 1988]. Strict integer, boolean, character and real arguments are passed to the function in evaluated form on a stack. And also strict tuple arguments are passed in head normal form on a stack [Smetsers et al. 1989].

Another way to do as little graph rewriting as possible is by changing the reduction order. This is possible in Clean by influencing the functional strategy by using annotations. With these annotations one can make the evaluation partially eager instead of lazy. If an evaluation is made eager it can often be executed faster. There are two types of annotations: global annotations and local annotations.

Global annotations change the reduction order for all applications of a particular function. These annotations are specified in the left hand side of a type definition of a rewrite rule, by putting a ! before the type of an argument. Such an annoted argument is always reduced to root normal form before the corresponding rule is applied. Note that such an annoted argument is always strict, and can therefore often be passed in evaluated form on a stack.

Local annotations change only the order of evaluation for a specific function application. Before the evaluation of the right hand side of a rule is continued following the functional strategy, all ! annoted nodes in the right hand side are evaluated. These annoted nodes are reduced in an arbitrary order.

A Clean program consists of modules. Modules can be compiled separately. There are two types of modules: definition modules and implementation modules. In general, each definition module has a corresponding implementation module. An exception is the main module, which consists of only an implementation module. An alternative kind of definition module is the system module, which allows the corresponding implementation module to be not a Clean program, and is used to implement delta-rules.

In a definition module graph rewrite rules can be specified in a so-called RULE-block. Types can be defined in a so-called TYPE-block. Type and rule definitions have as scope the implementation modules in which they are defined. They can also be exported if they are declared in the definition module, which makes it possible to import them in other modules. An entire module can be imported, but it is also possible to import only selected types and rules which were exported by the module.

2.2. The ABC machine.

In this section the ABC machine [Koopman et al. 1990] is described. First the ABC machine architecture is described, then an example is given of ABC code generated from a Clean program and finally the instruction set of the ABC machine is described.

The ABC machine is a sequential abstract machine designed to describe graph reduction. It is the target language for the compiler which translates the language Clean. Using the ABC machine we can describe

graph reduction on a level close to the level of a concrete machine. We can abstract from machine dependent issues, like addressing modes and register use. It also allows us to ignore some problems, like garbage collection and memory allocation, on the ABC machine level.

The ABC machine is a stack machine with 3 stacks, it does not have registers. The graph rewriting is performed in a heap (or graph store) in which nodes can be created and overwritten. Nodes which are no longer needed for the execution of the program are not explicitly deallocated. In an implementation on a concrete machine this deallocating should be done by a garbage collector.

In general, nodes consist of a descriptor, a code address and arguments. The descriptor indicates what kind of node it is. The code address (or evaluation address) is the address at which the code starts which reduces the node to head normal form. The arguments are node-id's of other nodes. There are special nodes for integers, characters, booleans, reals and strings. The argument of these nodes is not a node-id, but an integer, character, boolean, real or string. There is also an empty node.

The ABC machine uses 3 different stacks: the A-stack stores pointers to nodes, the B-stack stores non-pointers, like integers, booleans, characters and floating point values, and the C-stack pointers to code. (return addresses) There is a separate stack for pointers to nodes (the A-stack) to make garbage collection easier. The B-stack is used for computations using integers, booleans characters and floating point numbers. These computations can be done a lot faster on the B-stack than in the graph. So by using the B-stack as much as possible the program will execute much faster.

An ABC instruction consists of an instruction identifier and zero or more operands. The operands may be indices of one of the stacks, labels, descriptors, integers, reals, characters or strings. Operands are separated by spaces. An ABC program looks like an assembly language program, it is not block-structured. Every line consists of an optional label followed by a colon and an instruction or directive and eventually some comment, or just of a comment. Comment begins with '| |'.

For example, for the function `fac` in the Clean program in section 2.1 the following ABC code is generated by the Clean compiler [Smetsers 1989]: (see appendix A for more examples)

```
lFac:                                   || apply entry : 'Fac' node, argument n
                                           node and node to be overwritten by
                                           result on the A-stack
      pop_a          1                  || pop 'Fac' node from the A-stack
      jmp            m.1                || jump to label m1
nFac:                                   || node entry : 'Fac n' node on the A-
                                           stack
      push_args      0 1 1              || push argument n on the A-stack
m.1:
      set_entry      _cycle_in_spine 1  || store _cycle_in_spine as evaluation
                                           address to detect cycles in the spine
                                           of the graph
      jsr_eval                          || evaluate argument n
      pushI_a        0                  || push argument n on the B-stack
      pop_a          1                  || pop node of argument n from the A-stack
      jsr            sFac.1             || call strict entry of Fac to compute
                                           result on the B-stack
      fillI_b        0 0                || fill the result node with the integer
                                           result on the B-stack
      pop_b          1                  || pop the integer result from the B-stack
      rtn                               || return
sFac.1:                                 || strict entry : argument n (evaluated)
                                           on the B-stack
      eqI_b          +0 0               || n equal 0 ?
      jmp_true       m.2                || yes, jump to label m.2
      jmp            sFac.2             || no, jump to label sFac.2
m.2:
      pop_b          1                  || pop argument n from the B-stack
      pushI          +1                 || push 1 (result) on the B-stack
      rtn                               || return
sFac.2:
      push_b         0                  || push argument n on the B-stack
```

```
decI                                || subtract 1 from copy of n on the B-
                                       stack
jsr             sFac.1              || call strict entry of fac with argument
                                       n-1 on the B-stack to compute Fac (--I
                                       n)
push_b          1                   || push argument n on the B-stack
update_b        1 2                 || reorganize the B-stack
update_b        0 1
pop_b           1
mulI                                || compute *I n (Fac (--I n))
rtn                                 || return
```

2.2.1. ABC instructions for graph manipulation.

Below the instructions of the ABC machine are described. In these descriptions of the instructions I have used to following convention: node A-offset means the node of which the corresponding node-id is at position A-offset on the A-stack, to make the descriptions shorter and easier to understand.

Many ABC instructions refer to a elements on the stack using a position number or offset. The top element of a stack has offset zero, the second element of a stack has offset one, the third element has offset two, etc.

The instruction:

```
create
```

creates an empty node in the graph store, the node-id of this node is pushed on the A-stack.

The instruction:

```
fill            descriptor arity code_label A-offset
```

fills or overwrites node `A-offset`. The node is filled with the `descriptor`, address of the `code_label` and arguments. `Arity` is the number of arguments. The arguments are popped from the A-stack. The node-id on top of the A-stack is the first argument, the next node-id the second argument, etc.
There are also fill instructions to fill or overwrite nodes with integer, character, boolean, real and string nodes.

```
fill_a          A-offset1 A-offset2
```

fills or overwrites node `A-offset2` by a copy of node `A-offset1`.

```
set_entry       code_label A-offset
```

replaces the code address of node `A-offset` by the address of the `code_label`.

```
add_args        A-offset1 number_of_arguments A-offset2
```

fills or overwrites node `A-offset2` by a copy of node `A-offset1`, and then pops `number_of_arguments` arguments (node-id's) from the A-stack and adds these arguments to the node just filled or overwritten. The argument which is popped first (on top of the stack) is added as the first new argument, the argument which is popped second is added as the second new argument, etc.

```
del_args        A-offset1 number_of_arguments A-offset2
```

fills or overwrites node `A-offset2` by a copy of node `A-offset1`, and then pushes the last `number_of_arguments` arguments of the node just filled or overwritten on the A-stack and deletes these arguments from the node. The last argument of the node is pushed first, then the second last argument, etc.

2.2.2. ABC instructions for information retrieval from nodes.

```
push_args       A-offset arity number_of_arguments
```

pushes the first `number_of_arguments` arguments of node `A-offset` on the A-stack. `Arity` is the number of arguments of this node. The last argument to be pushed is pushed first, then the second last, etc.

There are also push argument(s) instructions, which push only one argument, or take the argument number or the number of arguments from the B-stack. And there are also replace arguments instructions, which also push the argument(s), but also pop the node-id of the node from the A-stack.

```
pushI_a          A-offset
```
pushes the integer which is stored in integer node `A-offset` on the B-stack. There are also instructions to push a boolean, character or real from a node on the B-stack.

```
get_node_arity A-offset
```
pushes the arity of node `A-offset`, i.e. the number of arguments of node `A-offset`, on the B-stack.

```
get_desc_arity A-offset
```
pushes the arity of the descriptor which is stored in node `A-offset`, i.e. the arity of the symbol which corresponds to the descriptor stored in node `A-offset`, on the B-stack.

```
push_ap_entry  A-offset
```
pushes the apply code address of the descriptor of node `A-offset` on the C-stack. The apply code address is the address of code which is used when evaluating curried functions. This address is stored in the descriptor.

2.2.3. ABC instructions for pattern matching.

```
eq_desc          descriptor number_of_arguments A-offset
```
pushes TRUE on the B-stack if the `descriptor` is equal to the descriptor of node `A-offset` and the `number_of_arguments` is equal to the number of arguments of node `A-offset`, otherwise FALSE.

```
eqI_a            integer_constant A-offset
```
compares the integer stored in node `A-offset` to the `integer_constant` and then pushes TRUE on the B-stack if they are equal, otherwise FALSE. There are also instructions to compare a boolean, character, real or string constant to a value in a node and then to push TRUE on the B-stack if they are equal, otherwise FALSE.

```
eqI_b            integer_constant B-offset
```
compares the integer at position `B-offset` on the B-stack to the `integer_constant` and then pushes TRUE on the B-stack if they are equal, otherwise FALSE. There are also instructions to compare a boolean, character, or real constant to a value on the B-stack and then to push TRUE on the B-stack if they are equal, otherwise FALSE.

2.2.4. ABC instructions to manipulate the A- and B-stack.

```
pop_a            number_of_elements
```
pops `number_of_elements` elements from the A-stack.

```
push_a           A-offset
```
pushes the element on the A-stack with offset `A-offset` on the A-stack, i.e. it creates a copy on top of the stack.

```
update_a         A-offset1 A-offset2
```
overwrites the element on the A-stack with offset `A-offset2` with the element on the A-stack with offset `A-offset1`.

The `pop_b`, `push_b`, and `update_b` instructions do the same with the B-stack as the `pop_a`, `push_a` and `update_a` instructions do with the A-stack.

2.2.5. ABC instructions to push constants on the B-stack.

```
pushI           integer_constant
```
pushes the `integer_constant` on the B-stack. There are also instructions to push character, boolean and real constants on the B-stack.

2.2.6. ABC instructions to change the flow of control.

```
jmp             label
```
jumps to the address of `label`, i.e. continues execution at the address of `label`.

```
jsr             label
```
calls the subroutine at the address of `label`, i.e. pushes the address of the instruction after this `jsr` instruction on the C-stack and continues execution at the address of the `label`.

```
jmp_false       label
```
pops a boolean from the B-stack, if this boolean is equal to FALSE execution continues at the address of the `label`, otherwise does nothing. (continues execution at the address of the instruction after this `jmp_false` instruction)

```
jmp_true        label
```
pops a boolean from the B-stack, if this boolean is equal to TRUE execution continues at the address of the `label`, otherwise does nothing. (continues execution at the address of the instruction after this `jmp_true` instruction)

```
rtn
```
returns from a subroutine, i.e. pops an address from the C-stack and continues execution at this address.

```
jsr_eval
```
pushes the address of the instruction after this `jsr_eval` instruction on the C-stack and jumps to the code address of the node of which the node-id is on top of the A-stack.

```
jmp_eval
```
jumps to the code address of the node of which the node-id is on top of the A-stack.

```
halt
```
ends the execution of the program.

```
dump            string
```
prints the `string` and ends the execution of the program.

2.2.7. ABC instructions to generate output.

```
print           string
```
prints the `string`.

```
print_symbol    A-offset
```
prints a string representation of node `A-offset`.

2.2.8. ABC instructions to implement delta-rules.

Some examples are:

```
addI            || pop two integers from the B-stack, add them, and push the
                   result on the B-stack.
mulI            || pop two integers from the B-stack, multiply them, and push
                   the result on the B-stack.
subI            || pop two integers from the B-stack, subtract the second
                   popped integer from the other integer, and push the result
                   on the B-stack.
incI            || increment the integer on top of the B-stack.
ltI             || pop two integers from the B-stack, compare them, if the
                   first popped integer is less then the other integer push
                   TRUE on the B-stack, otherwise FALSE.
```

There are many more instructions, also instructions which operate on booleans, characters, reals and strings.

2.3. The MC68020 microprocessor.

To be able to construct a code generator it is necessary to know the target processor. In this section I will describe the relevant parts of the architecture and instruction set of the Motorola MC68020 processor. Interrupts, privileged instructions, etc. are not discussed. An example of MC68020 code generated for a small Clean program by the Clean compiler and the code generator described in this paper is given. Execution speed of instructions is also discussed. This information was taken from [Motorola 1985] and is used to conclude what kind of code has to be generated.

2.3.1. The MC68020 registers.

The MC68020 is a 32-bit register machine. It has:
- eight 32-bit data registers, called D0, D1, .., D7
- eight 32-bit address registers, called A0, A1, ..., A7.
- a 32-bit program counter, called PC.
- a 16-bit status register, called SR.

Registers D0-D7 are used as data registers for byte, word, long word and quad word operations. Registers A0-A6 are address registers that may be used as software stackpointers and base address registers. Register A7, also called SP, is the stack pointer. The address registers may also be used for word and long word operations. All of the 16 (address and data) registers may be used as index registers.

The status register (SR) contains among other things the condition codes: extend (X), negative (N), zero (Z), overflow (V), and carry (C).

2.3.2. The MC68020 data types.

Seven basic data types are supported:
- Byte integers (8 bits).
- Word integers (16 bits).
- Long word integers (32 bits).
- Quad word integers (64 bits).
- Bits.
- Bit fields (Field of 1-32 consecutive bits).
- Binary Coded Decimal (BCD) Digits (Packed: 2 digits/byte. Unpacked: 1 digit/byte).

In addition operations on memory addresses (32 bits) are supported.

The ABC machine uses bytes, words, long words and memory addresses. Elements of the B-stack are long words, elements of the A-stack and C-stack are memory addresses, bytes are used in string and boolean operations and descriptors are words. But the ABC machine does not have instructions using quad word integers, bits, bit fields and BCD digits.

2.3.3. The MC68020 addressing modes.

There are 18 addressing modes:

Name:	Assembler syntax:	Result (in C):
Data register direct	Dn	Dn
Address register direct	An	An
Address register indirect	(An)	*An
Address register indirect with postincrement	(An)+	*An++
Address register indirect with predecrement	-(An)	*--An
Address register indirect with displacement	(d16,An) or d16(An)	*(d16+An)
Address register indirect with index (8-bit displacement)	(d8,An,Xn)	*(d8+An+Xn)
Address register indirect with index (base displacement)	(bd,An,Xn)	*(d16+An+Xn)
Memory indirect post-indexed	([bd,An],Xn,od)	*(*(bd+An)+Xn+od)
Memory indirect pre-indexed	([bd,An,Xn],od)	*(*(bd+An+Xn)+od)
Program counter indirect with displacement	(d16,PC)	*(d16+PC)
PC indirect with index (8-bit displacement)	(d8,PC,Xn)	*(d8+PC+Xn)
PC indirect with index (base displacement)	(bd,PC,Xn)	*(bd+PC+Xn)
PC memory indirect post-indexed	([bd,PC],Xn,od)	*(*(bd+PC)+Xn+od)
PC memory indirect pre-indexed	([bd,PC,Xn],od)	*(*(bd+PC+Xn)+od)
Absolute short	a16.W	*a16
Absolute long	a32.L	*a32
Immediate	#data	data

where:

Dn = Data register, D0-D7.
An = Address register, A0-A7.
d8 = 8 bit signed displacement.
d16 = 16 bit signed displacement.
Xn = address or data registers used as an index register, form is Xn.SIZE*SCALE, where:
SIZE is W or L (indicates that the index register is a signed word (W) or long word (L)).
SCALE is 1, 2, 4 or 8 (the value in the index register is multiplied by SCALE).
bd = a signed 16 or 32 bit displacement.
od = a signed 16 or 32 bit displacement.
data = an 8, 16 or 32 bit integer.

a16 = a signed 16 bit address.
a32 = a 32 bit address.

For the complicated addressing modes: (bd,An,Xn), ([bd,An],Xn,od), ([bd,An,Xn],od), (bd,PC,Xn), ([bd,PC],Xn,od) and ([bd,PC,Xn],od) all of the operands (bd, od, An and Xn) are optional, if an operand is left out, the value of the operand is assumed to be zero.

Because the ABC machine does not have arrays, we will not use the indexing addressing modes. But because the index registers are optional in some index addressing modes, they may still be useful without index register. By leaving out the index registers we get two addressing modes: ([bd,An],od) and ([bd,PC],od). For both addressing modes the addressed value can also be addressed by two register indirect addressing modes. For example, instead of:

```
ADD.L     ([8,A0],4),D0
```

we can use:

```
MOVEA.L   8(A0),A1
ADD.L     4(A1),D0
```

Strangely enough the two register indirect with displacement instructions are executed faster than the one memory indirect instruction. Therefore, and also because the length of both instruction sequences (the number of instruction words) is the same, it is better not to use these addressing modes. Although sometimes an additional address register (A1 in the example) is necessary.
For these addressing modes (([bd,An],od) and ([bd,PC],od)) the displacements (bd and od) are optional. If one or two of these displacements are left out, the execution times of two register indirect (possibly with displacement) instructions are still better and the instruction lengths are still the same. So then it is also better to use two register indirect (possibly with displacement) instructions.
But in the case of memory indirect addressing, the displacement(s) may be long words, and for the register indirect with displacement addressing mode the displacements may not be a long word. But we will never have to use long word displacements. So we will never use the memory indirect addressing mode.

We will also not use the absolute short and absolute long addressing modes, because on the Macintosh we can only use them to access the Macintosh system variables, not to access application variables or code. This is because the operating system may load a program anywhere in memory, but can not calculate the absolute addresses. But on other machines the absolute addressing modes may be useful.

Therefore the addressing modes we will use in the code generator are:

Name:	Assembler syntax:	Result (in C):
Data register direct	Dn	Dn
Address register direct	An	An
Address register indirect	(An)	*An
Address register indirect with postincrement	(An)+	*An++
Address register indirect with predecrement	-(An)	*--An
Address register indirect with displacement	(d16,An) or d16(An)	*(d16+An)
Program counter indirect with displacement	(d16,PC)	*(d16+PC)
Immediate	#data	data

2.3.4. The MC68020 instruction set.

Below the instructions of the MC68020 are described which are useful for implementing the ABC machine. If an operation can be performed in several ways using one MC68020 instruction, it is described which instruction is the most efficient.

The MOVE instruction.

MOVE <sea>,<dea> moves (copies) <sea> to <dea>. The source effective address (<sea>) may be any addressing mode. The destination effective address (<dea>) may not be an immediate value or use the program counter, for example (d16,PC) is not allowed. All operand sizes (byte, word and long) are allowed. In MC68020 assembler these sizes are expressed with a .B (byte), .W (word) or .L (long) after the instruction, thus MOVE.B, MOVE.W or MOVE.L. Without such an extension the operand(s) are assumed to be words.
If the <dea> is an address register, the instruction is called MOVEA <sea>,An. For MOVEA operand size byte is not allowed. If the operand size is word, the word is sign extended to a long word and this long word is stored in the address register.
Instead of using MOVE.L #i,Dn it is better to use MOVEQ #i,Dn if i is between -128 and 127. This MOVEQ instruction is faster and the instruction consists of only 1 word, instead of 3 words.
Also MOVE #0,<ea> can be replaced by the faster and shorter CLR <ea>. For CLR all operand sizes are allowed. CLR An is not possible, so this improvement is not possible for MOVEA, but instead we can subtract An from itself, resulting in zero, with SUB.L An,An. (Instruction SUB is described below)

The other data movement instructions.

With MOVEM registerlist,<ea> selected registers are stored at consecutive memory locations starting at the location specified by <ea>. Address register indirect with postincrement, register direct, immediate, and addressing modes using the program counter are not allowed for <ea>.
With MOVEM <ea>,registerlist selected registers are loaded from consecutive memory locations starting at the location specified by <ea>. Address register indirect with predecrement, register direct and immediate addressing modes are not allowed for <ea>.
The registers are stored in memory in the order from D0 to D7, and then from A0 to A7. The operand size may be word or long word.

The LEA <ea>,An instruction loads the address of the object addressed by <ea> (such an address is called an *effective address* by Motorola) in register An. The operand size is always long and for <ea> register direct, immediate, postincrement and predecrement addressing modes are not allowed.

The PEA <ea> instruction pushes the effective address <ea> onto the stack. The operand size is always long and the same addressing modes are allowed as for LEA.

The EXG Rn,Rm instruction exchanges the contents of registers Rn and Rm. For Rn and Rm data registers and address registers are allowed. (exchanging an address register and a data register is possible) The operand size is always long.

The ADD and SUB instructions.

ADD <sea>,<dea> adds the <sea> to the <dea>. The <sea> or the <dea> (at least one) has to be a data register or an immediate value. Immediate and PC relative addressing modes are not allowed for the <dea>, if the <dea> is an address register, the instruction is called ADDA. (see below) All operand sizes are allowed. If the <sea> is an immediate value, the instruction is called ADDI.
For all operand sizes ADD #n,Dn can be replaced by the faster and shorter ADDQ #n,Dn if n is between 1 and 8.

ADDA <sea>,An adds the <sea> to An. Operand sizes word and long are allowed, if the operand size is word, the <sea> word is sign extend to long and then added to the whole (long) address register.
Again ADDA #n,An can be replaced by the faster and shorter ADDQ #n,An if n is between 1 and 8.

To add a 16 bit signed immediate value n (not between 1 and 8) to an address register, it is faster and shorter to use ADDA.W #n,An or LEA n(An),An than ADDA.L #n,An.

For SUB, SUBI, SUBQ and SUBA the same addressing modes are allowed as for ADD, ADDI, ADDQ and ADDA, but now the <sea> is subtracted from the <dea>.

The CMP and TST instructions.

CMP <ea>,Dn compares the <ea> to Dn, i.e. it subtracts the <ea> from the value in Dn, but does not store the result in Dn. All operand sizes are allowed.

CMPA <ea>,An compares the <ea> to An. Operand sizes word and long word are allowed. If the operand size is word, the <ea> signed word is sign extended to long and compared to the whole (long) address register An. To compare An with a signed 16 bit integer n, it is faster and shorter to use CMPA.W #n,An then CMPA.L #n,An.

CMPI #n,<ea> compares the immediate value n with <ea>. For <ea> immediate and address register direct addressing modes are not allowed. All operand sizes are allowed.
For all operand sizes CMP #0,<ea> can be replaced by the faster and shorter TST <ea>.

The MULS and DIVS instructions.

MULS.L <ea>,Dn multiplies the signed long word in Dn by the signed long word <ea>.

DIVS.L <ea>,Dn divides the signed long word in Dn by the signed long word <ea>.
DIVSL.L <ea>,Dr:Dq divides the signed long word in Dq by the signed long word <ea>, the remainder is stored in Dr. There is no instruction to calculate the remainder without also calculating the quotient.
For all three instructions <ea> can not be an address register. The MC68020 does have other multiplication and division instructions, but we will not use them.

The EXT instruction.

EXT.W Dn sign extends the byte in Dn to a word. (bit 7 is copied to bits 8-15)
EXT.L Dn sign extends the word in Dn to a long. (bit 15 is copied to bits 16-31)
EXTB.L Dn sign extend the byte in Dn to a long. (bit 7 is copied to bits 8-31)

The NEG and NOT instructions.

NEG <ea> negates (subtracts from zero) the <ea>.
NOT <ea> complements (changes all bits, or subtracts from -1) the <ea>.
For both instructions address register direct, immediate and PC relative addressing modes are not allowed for <ea>, all operand sizes are allowed.

The AND and OR instructions.

AND <sea>,<dea> ands the <sea> to the <dea>. The <sea> or the <dea> (at least one) has to be a data register or an immediate value. Both the <dea> and the <sea> may not be an address register, for the <dea> immediate and PC relative addressing modes are also not allowed. All operand size are allowed. If the <sea> is an immediate value, the instruction is called ANDI.
For OR and ORI the same addressing modes are allowed as for AND and ANDI, but now the <sea> is ored to the <dea>.

The EOR instruction.

`EOR <sea>,<dea>` exclusive-ors the `<sea>` to the `<dea>`. The `<sea>` has to be a data register or an immediate value. For the `<dea>` address registers direct, immediate and PC relative addressing modes are not allowed. All operand size are allowed. If the `<sea>` is an immediate value, the instruction is called `EORI`.

The shift instructions.

`LSL #n,Dy` shifts data register `Dy` `n` bits to the left, `n` must be between 1 and 8.

`LSL Dx,Dy` shifts data register `Dy` `Dx` (modulo 64) bits to the left.
Zero bits are shifted in from the right. The operand size is always long. The MC68020 can also shift a word in memory by one bit, but these instructions will not be used.

For `LSR`, `ASL` and `ASR` the same addressing modes are possible as for `LSL`.
`ASL` also shifts left, but does not clear the overflow flag, but sets it if the sign bit (most significant bit) changes during the shift, otherwise clears it. Because `LSL` is faster than `ASL` and because we don't need to know whether or not an overflow occurs, we will always use `LSL`.
`LSR` shifts to the right. Zero bits are shifted in from the left.
`ASR` shifts to the right. Sign bits (most significant bit) are shifted in from the left, so that the sign does not change during the shift. `LSR` is faster than `ASR`, but the result is not the same for negative integers.

Unconditional branch instructions.

`BRA <label>` branches to (continues execution at) the `<label>`, i.e. the address of the `<label>` is loaded into the PC (Program Counter). This address is specified by a displacement from the PC. And this address is calculated by adding the PC+2 (i.e. the value in PC after fetching the first instruction word) and a displacement. This displacement may be a signed byte, word or long. For small displacements the instruction is shorter, but just as fast as for long displacements.

`BSR <label>` calls the subroutine at `<label>`,i.e. it does the same as `BRA <label>`, but first pushes the address of the instruction after this (`BSR`) instruction onto the system stack by subtracting 4 from SP and moving the address to (SP). Again the instruction is shorter for small displacements, but just as fast.

`JMP <ea>` jumps (continues execution at) the address of `<ea>`, i.e. the address of `<ea>` is loaded into PC. For `<ea>` the following addressing modes are not allowed: (data and address) register direct, address register with post- or pre-increment and immediate. `JMP d16(PC)` can be replaced by a `BRA` instruction, this is faster and just as long, or shorter if the displacement is a signed byte.

`JSR <ea>` calls the subroutine at address `<ea>`, i.e. it does the same as `JMP <ea>`, but first pushes the address of the instruction after this (`JSR`) instruction onto the system stack by subtracting 4 from SP and moving the address to (SP). The same addressing modes are allowed as for `JMP`. `JSR d16(PC)` can be replaced by a `BSR` instruction, this is faster and just as long, or shorter if the displacement is a signed byte.

`RTS` returns from subroutine, i.e. (SP) is moved to PC and 4 is added to SP.

Conditional instructions.

`Bcc <label>` branches to the `<label>` if the condition `cc` is true. `cc` can be any condition described below. Just like for `BRA` the label is specified by a displacement which is added to PC, and the displacement may be a signed byte, word or long. For small displacements the instruction is shorter.

If the branch is taken (if the condition is true), the instruction is just as fast for short displacements as for long displacements. But if the branch is not taken, the instruction is a little bit faster for shorter displacements. For byte displacements a not taken branch is faster than a taken branch, for word displacements the difference is small, and for long displacements there is no difference in speed.

`DBcc Dn,<label>` tests the condition `cc` and then if the condition is false, decrements (subtracts 1 from) `Dn` and then if `Dn` is not equal to -1 branches to the `<label>`. The displacement which is added to PC must be a word.

`Scc <ea>` sets all bits of the `<ea>` to one (the byte to -1) if the condition `cc` is true, otherwise to zero. The operand size is always byte. Address register direct, immediate and PC-relative addressing modes are not allowed for `<ea>`.

The condition codes.

The possible condition codes are:

`EQ`	equal (=)
`NE`	not equal (!=)
`GT`	greater than (signed >)
`GE`	greater or equal (signed ≥)
`LE`	less or equal (signed ≤)
`LT`	less than (signed <)
`HI`	high (unsigned >)
`CC` or `HS`	carry clear or high or same (unsigned ≥)
`LS`	low or same (unsigned ≤)
`CS` or `LO`	carry set or low (unsigned <)
`PL`	plus (≥0)
`MI`	minus (<0)
`VC`	overflow clear
`VS`	overflow set
`T`	true
`F`	false

`Bcc` after `CMP <sea>,<dea>` branches if `<dea> cc <sea>`, for example

```
       CMP.L      #1000,D0
       BGT        label
```

branches if `D0 GT #1000` (`D0` greater than 1000).

Of the instructions described above the following instructions set the condition codes: (some don't set the X flag, which can be used for multiprecision and mixed size arithmetic, but none of the conditions and instructions described above depends on this flag)

```
MOVE,      MOVEQ,     CLR,       ADD,       ADDI,      SUB,       SUBI,      CMP,
CMPA,      CMPI,      TST,       MULS,      DIVS,      DIVSL,     EXT,       EXTB,
NEG,       NOT,       AND,       ANDI,      OR,        ORI,       EOR,       EORI,
LSL,       ASL,       LSR,       ASR
```

and `ADDQ` and `SUBQ` to a data register.

Note that the following instructions don't set the condition codes:

```
MOVEA,     ADDA,      SUBA,      MOVEM,     LEA,       PEA,       EXG,       Scc
```

and `ADDQ` and `SUBQ` to an address register.

2.3.5. Example of MC68020 code of a Clean function.

The MC68020 which is generated by the ABC compiler described in this paper from the ABC code in section 2.2 for the function fac is: (see appendix A for more examples)

```
lFac:                                   ; apply entry : 'Fac' node in register A2,
                                          argument n node in register A1 and node
                                          to be overwritten by result on the A-
                                          stack in register A0
       BRA       m.1                    ; jump to label m.1
nFac:                                   ; node entry : 'Fac n' node in register A0
       MOVEA.L   8(A0),A1               ; move pointer to argument part of node to
                                          A1
       MOVEA.L   (A1),A1                ; move argument n to A1
m.1:   LEA       _cycle_in_spine,A2     ; load address of _cycle_in_spine code in
                                          A2
       MOVE.L    A2,(A0)                ; store _cycle_in_spine as evaluation
                                          address to detect cycles in the spine of
                                          the graph
       MOVE.L    (A1),D6                ; load reduction address of n
       BEQ       eval_0                 ; branch if reduction address = _hnf (0)
       MOVE.L    A0,-(A3)               ; save register A0
       MOVEA.L   A1,A0                  ; move argument n node to A0
       MOVEA.L   D6,A1                  ; move evaluation address of n to address
                                          register
       JSR       (A1)                   ; evaluate argument n node
       MOVEA.L   A0,A1                  ; move address of evaluated argument n
                                          node to A1
       MOVEA.L   (A3)+,A0               ; load register A0
eval_0:
       MOVE.L    A0,-(A3)               ; push node to be overwritten by result on
                                          the A-stack
       MOVE.L    8(A1),D0               ; load evaluated integer n in D0
       BSR       sFac.1                 ; call strict entry of Fac to compute Fac
                                          n in D0
       MOVEA.L   (A3)+,A0               ; load address of node to be overwritten
                                          by result in A0
       MOVE.L    A0,D1                  ; copy address of result node to D1
       CLR.L     (A0)+                  ; overwrite node to store result, first
                                          store evaluation address,
       MOVE.L    #!INT+0,(A0)+          ; then the descriptor of an integer node,
       MOVE.L    D0,(A0)                ; and finally the integer in D0 (the
                                          result of Fac n)
       MOVEA.L   D1,A0                  ; return the address of the result node in
                                          A0
       RTS                              ; return, with result in A0
sFac.1:
       TST.L     D0                     ; n equal 0 ?
       BNE       sFac.2                 ; no, jump to label sFac.s
m.2:
       MOVEQ     #1,D0                  ; return 1 in D0
       RTS                              ; return
sFac.2:
       MOVE.L    D0,(A4)+               ; push n on the B-stack
       SUBQ.L    #1,D0                  ; subtract one from n in D0
       BSR       sFac.1                 ; call strict entry of Fac to compute Fac
                                          (--I n) in D0
       MULS.L    -(A4),D0               ; multiply result by n to compute Fac n
       RTS                              ; return, with result in D0
```

2.3.6. The MC68020 cache.

To improve the performance of the processor, the MC68020 has an instruction cache. During the execution of an instruction the processor tries to prefetch the following words in the instruction stream. The cache is used to store these prefetched words.

The cache contains a high speed memory of 64 long words. Every time an instruction fetch occurs, the processor first checks if the word required is already in the cache. If it is, the word does not have to be fetched from memory. Otherwise the long word, which contains this word, is fetched from memory and stored in the cache.

This cache speeds up small loops and small recursive functions, because instructions in the loop have to be fetched from memory only once.
Only instructions are stored in the cache, other memory accesses don't use the cache.

2.3.7. MC68020 instruction execution timing.

The number of memory accesses to fetch or write a word or long word depends on the size of the data bus. For the MC68020 the data bus can be 8, 16 or 32 bits wide, I will assume it is 32 bits wide, because it usually is (also on the Mac II). Then the number of memory accesses to fetch or write:
- a byte is 1.
- a word is 1, if the word is word aligned (at an even address), otherwise 2.
- a long word is 1, if the long word is long word aligned (at an address which can be divided by 4), otherwise 2.

If memory can be accessed without wait states, every memory access costs 3 clock cycles. Then accesses to words which are not word aligned and long words which are not long word aligned cost 3 clock cycles extra. Therefore it is important to align words and long words.

The number of memory accesses necessary to fetch an instruction depends on the length of the instruction, whether or not the instruction is long word aligned, and on the contents of the cache. (see also section 2.3.6) Nearly all instructions consist of one word (some two words), and 0-4 effective address extension words. The number of effective address extension words depends on the addressing modes which are used in the instruction. (see table below)

Exact instruction timing calculations for the MC68020 are difficult because of:
- an instruction cache and instruction prefetch.
- operand misalignment.
- instruction execution overlap.

Therefore [Motorola 1985] only gives execution times for a machine with 32-bit databus, no wait state memory and long word aligned operands and stack. For every instruction it gives 3 execution times: the best case, the cache case and the worst case. Cache case assumes the instruction is already in the cache, so that no extra clock cycles are necessary to fetch the instruction, and that no execution overlap occurs with other instructions. Best case execution times are often better than cache case due to execution overlap with other instructions. And worst case execution times are often worse than cache case because the instruction has to be fetched. I will use these cache case instruction times because if the cache is effective, this will usually be the most accurate time.

In the following table for the addressing modes used by the code generator the following is indicated:
- the fetch effective address (FEA) time in clock cycles.
- the calculate effective address (CEA) time in clock cycles.
- the fetch immediate word effective address (FIWEA) time in clock cycles.
- the fetch immediate long effective address (FILEA) time in clock cycles.
- the calculate immediate word effective address (CIWEA) time in clock cycle.
- the number of effective address extension words.

These times can be used to calculate the cache case execution time together with the second table below.

Addressing mode:	FEA time	CEA time	FIWEA time	FILEA time	CIWEA time	extension words
`Dn`	0	0	2	4	2	0
`An`	0	0	-	-	-	0
`(An)`	4	2	4	4	2	0
`(An)+`	4	2	6	8	4	0
`-(An)`	5	2	5	7	?	0
`(d16,An)`	5	2	5	7	4	1
`(d16,PC)`	5	2	-	-	-	1
`#<data>.B`	2	-	-	-	-	1
`#<data>.W`	2	-	-	-	-	1
`#<data>.L`	4	-	-	-	-	2

The number of clock cycles needed to execute the following instruction are: (the number of clock cycles for the other instructions can be found in appendix B)
(`Rn` means `An` or `Dn`, `<mea>` means memory effective address, not a register)

`MOVEQ #<data>,Dn`	2
`ADDQ #<data>,Rn`	2
`SUBQ #<data>,Rn`	2
`EXG Ry,Rx`	2
`MOVEM <ea>,register list`	8 + CIWEA time + 4 * number of registers
`MOVEM register list,<ea>`	4 + CIWEA time + 3 * number of registers
`ADD <ea>,Dn`	2 + FEA time
`ADDA <ea>,An`	2 + FEA time
`SUB <ea>,Dn`	2 + FEA time
`SUBA <ea>,An`	2 + FEA time
`CMP <ea>,Dn`	2 + FEA time
`CMPA <ea>,An`	4 + FEA time
`LEA <ea>,An`	2 + CEA time
`MULS.L`	43 + FIWEA time
`DIVS.L`	90 + FIWEA time
`TST <ea>`	2 + FEA time
`CLR Dn`	2
`CLR <mea>`	4 + CEA time.
`BRA`	6
`BSR`	7
`JMP (d16,PC)`	? (probably 8, same as `JMP (d16,An)`)
`JSR (d16,PC)`	? (probably 9, same as `JMP (d16,An)`)
`LSL #n,Dy`	4
`LSL Dx,Dy`	6
`ASL #n,Dy`	8
`ASL Dx,Dy`	8

The number of clock cycles needed to execute the `MOVE` and `MOVEA` instructions are:

	Destination:		
Source	`An` or `Dn`	`(An)` or `(An)+`	`-(An)` or `(d16,An)`
`An` or `Dn`	2	4	5
`(An)` or `(An)+`	6	7	7
`-(An)` or `(d16,An)` or `(d16,PC)`	7	8	8
`#<data>.B`, `#<data>.W`	4	6	7
`#<data>.L`	6	8	9

Using these tables we can conclude that:

- Access to registers is a lot faster than memory access. For example to add a value in a register to another register costs 2 clock cycles, while adding a value in memory using (An) to a register costs 6 clock cycles.

- Predecrementing (-(An)) or postincrementing ((An)+) an address register before addressing is not much slower than without predecrement or postincrement. For example `MOVE.L (A0)+,D0` is just as fast as `MOVE.L (A0),D0` and `MOVE.L -(A0),D0` takes only 1 clock cycle more.

- Using a 16 bit displacement ((d16,An)) is often only 1 clock cycle more expensive than not using a displacement ((An)). But because the instruction is one word longer, the difference is bigger when the instruction is not yet in the cache.

- The `MOVEQ`, `ADDQ` and `SUBQ` to a register are three times as fast and short than the normal `MOVE`, `ADD` and `SUB` instructions. `ADDQ` and `SUBQ` to a value in memory are also faster and shorter.

- For long word values between -128 and 127 it is often faster and shorter first to move the value in a data register using `MOVEQ` and then do the operation, instead of doing the operation with an immediate long addressing mode. For example instead of:

```
CMPI.L     #100,D0
```

it is faster and shorter to use:

```
MOVEQ      #100,D1
CMP.L      D1,D0
```

The first instruction sequence costs 6 clock cycles, while the second costs 4 clock cycles.

- `CLR <ea>` is faster than `MOVE #0,<ea>` and `TST <ea>` is faster than `CMPI #0,<ea>`.

- `CMP <ea>,Dn` is faster than `CMPA <ea>,An`, but for `MOVE`, `ADD` and `SUB` there is no difference in speed between data and address registers.

- `BRA` is faster than `JMP (d16,PC)`, and `BSR` is faster than `JSR (d16,PC)`.

- The multiply and division instructions are very slow compared to the other instructions. Multiplying a data registers by another data register costs 45 clock cycles, such a division costs 92 clock cycles, but such an an addition or subtraction costs just 2 clock cycles.

- `LSL` is faster than `ASL`.

3. Representing the ABC machine.

In this chapter is described how the data structures (heap, stacks etc.) used by the ABC machine [Koopman et al. 1990] are represented on a register machine, and then how they are represented on the MC68020 [Motorola 1985]. How nodes are represented in the heap, and why this representation was chosen, is described. Also the representation of strings is discussed.

3.1. The stacks.

The A, B and C stack of the ABC machine have to be stored in memory on a register machine. The best representation of these 3 stacks is a contiguous area of memory for each of them, with a pointer to the top of the stack. (stackpointer)
Because these stacks are used very often these stackpointers should be stored in registers rather than in memory. When pushing and popping values on and off a stack, the push and pop instructions of the machine can be used, so that values can be pushed or popped with just one instruction.

3.2 The heap.

The heap is a contiguous area of memory in which nodes are allocated. Because very often space is allocated on the heap, the pointer to the top of the heap is stored in a register and the number of free cells in the heap also in a register.

3.3. Representing nodes.

The representation of nodes should be so that:
- values stored in the node can be accessed fast.
- a node can be overwritten fast and by a larger node.
- garbage collection can be done fast.

Overwriting a small node by a large node causes problems, because no memory is available for the extra arguments. This problem is usually solved by creating a new (large) node and overwriting the small node by an indirection node, which points to the large node just created. But this makes access to nodes more expensive, because sometimes a pointer chain has to be traversed. Also the overwriting is more expensive, because not only the new node has to be created, but also an indirection node. Therefore this approach was not chosen.

Instead, just as for the previous ABC compiler (for the Sun) [Weijers 1990], the node was split into two parts: a fixed size part and a variable size part [Nöcker et al. 1991]. The fixed size part contains those fields that must be present in every node and a pointer to the variable size part. The variable size part contains the arguments of the node. Using this representation a small node can be overwritten by a large node by creating a new variable size part containing the arguments and overwriting the fixed size part of the small node with the other fields of the large node and a pointer to the variable size part just created.

The advantages of this representation are:
- It is never necessary to traverse a pointer chain to access a node.
- If a node is overwritten by a larger node, creating the node is less expensive in time, because it is now not necessary to create an indirection node.
- No memory is used for indirection nodes.

- Arguments can be shared between nodes, so that nodes with more than one argument can be copied faster, because the arguments don't have to be copied, but just the fixed size part. But this has as disadvantage that, if a node is overwritten by a node which is smaller or of the same size, the arguments of the node may not be overwritten. So that a new variable size part has to be allocated in the heap.

The disadvantaged of this representation are:
- One extra memory access is necessary to access the arguments of a node.
- Nodes are a little bit larger, because they now also contain a pointer to the variable size part.
- Because nodes are larger, creating a node is more expensive in time, except when a node is overwritten by a larger node. (see above)

Basic value nodes, i.e. integer, character, boolean, real and string nodes, don't have arguments, so the pointer to the variable size part doesn't have to be stored in the fixed size part. If the value to be stored in the basic value node (e.g. 3 for an integer 3 node) can be stored in fewer or the same number of bytes than a pointer, no variable size part is necessary. Because in that case the memory location(s) in the fixed size part, normally occupied by the pointer to the variable size part, is used to store this value. This can usually be done for integer, character and boolean values. Because there is no variable size part in such a case, the disadvantages described above don't apply to these nodes. But if the value to be stored in the basic value node can not be stored in the number of bytes necessary to store a pointer, then a variable size part is used to store this value. And a pointer to this variable part is stored in the memory location(s) in the fixed size part normally occupied by the pointer to the variable size part containing the arguments. Usually this is necessary for real and string values.

The other information (beside the arguments) we have to be able to retrieve from a node is:

- the address of the reduction code.
  If the node is not (yet) in head normal form, this location is usually accessed three times: first it is used to reduce the node, then it is overwritten to detect cycles in the spine of the graph and then it is overwritten to prevent reducing it again. After the node has been reduced this location may be accessed many more times. So it is important that this location can be accessed very fast.

- the symbol of the node. (for example INT or CONS)
  This symbol is used during pattern matching and access should therefore be very fast.

- the address of the apply reduction code.
  This address is used to reduce an APply of a node. (for curried applications) It is not used often and therefore access doesn't have to be fast.

- the number of arguments of the node.
  This information is necessary for the garbage collector and some ABC instructions which are not executed very often, for example `add_args`, `del_args` and `get_node_arity`. Because the garbage collector uses this information access has to be fast, but doesn't have to be as fast as access to the address of the reduction code and the symbol of the node.

- the arity of the symbol.
  This information is necessary for the ABC `get_desc_arity` instruction, which is not used very often. Access doesn't have to be fast.

- the string representation of the name of the symbol. (for example "INT" or "CONS")
  This string is used to print nodes with the ABC `print_symbol` instruction. Access doesn't have to be fast.

Access to the address of the reduction code and the symbol of the node should be very fast, therefore they should be stores in the (fixed size part of) the node.

The address of the apply reduction code, the arity of the symbol and the string representation of the name of the symbol only depend on the symbol. For every symbol this information could be stored in a record. Every node should then contain a pointer to the record of the symbol of the node. Then to access this information an extra memory access is necessary, but this is no problem, because this information doesn't

have to be accessed fast. Using such a pointer is faster than storing all the information in the node, because the pointer is smaller than the record, and therefore the nodes will be smaller and can be created faster. This record is called the *symbol record*.

Only the number of arguments has not yet been stored. We don't want to store this in the node, because the number of arguments is not accessed very often and this would make the nodes larger. We can't store the number of arguments in the symbol record, because nodes with the same symbol can have a different number of arguments. To solve this we could make a record for every number of arguments, but this would cost to much memory.

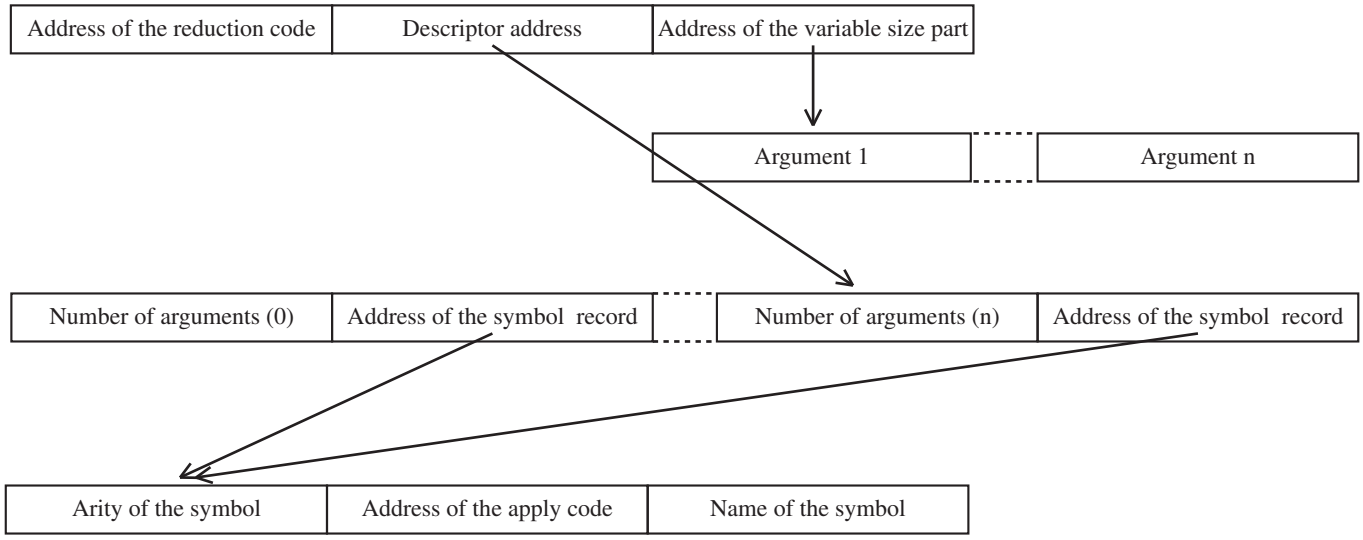
A better solution is to make (for every symbol) an array with an element for every number of arguments. Every element of the array contains the number of arguments and a pointer to the start of the corresponding symbol record. (the same for every element of an array) The first element of the array is for 0 arguments, the second element for 1 argument, etc. The elements are called *descriptor elements*.

Now the pointer in the node which points to the symbol record should be replaced by a pointer to the array element (the descriptor element) which contains the number of arguments of the node and a pointer to the symbol record. This of course means that access to the symbol record now requires an additional memory access, but this is not a problem, because the information in the symbol record doesn't have to be accessed fast. The address of a descriptor element will be called a descriptor address. Note that when the number of arguments of a node is changed, for example with `add_args` or `del_args`, the address of the array element can be calculated by adding the size of an element * (number of arguments - previous number of arguments) to the old address.

The symbol of the node is only used during pattern matching. Because we always know the number of arguments of a node during pattern matching, we can just as well use the address of the array element in the node for pattern matching. Then we don't have to store the symbol of the node in the node.

Consequently, we use the following data structures:

1. The fixed size part of a node consists of:
   - the address of the reduction code.
   - the descriptor address.
   - the address of the variable size part of the node, which contains the arguments, or a basic value for a basic value node. For a basic value node for which the size of a basic value is smaller than or the same as the size of a pointer, the basic value is stored in this location.
2. For every symbol there exists an array of [0 .. arity] elements. Element number n consists of:
   - the number of arguments. (n)
   - a pointer to the symbol record of this symbol.
3. For every symbol there exists a symbol record consisting of:
   - the arity of the symbol.
   - the address of the apply reduction code.
   - the string representation of the name of the symbol.



If the register machine efficiently supports address arithmetic, the array can be represented more compact. (see section 3.5)

3.4. Representing strings.

Possible string representations are:
- An array of characters which ends with a special character to indicate the end of the string.
- The length of the string and an array of characters.
- A list of characters.
- More complex representations, for example a list of arrays of characters.

A list of characters is not a good representation, because it uses too much memory. For every character of the string not only memory has to be allocated for the character, but also for a pointer, which usually occupies 4 times as much memory as a character.

More complex representations could result in faster execution than the other representations, but are difficult to program and often use too much memory. It is probably not worth while to spend a lot of time on designing and programming a complex string representation, because for most programs strings are not very important.
The other two string representations don't use a lot of memory.

Using a special character to indicate the end of the string has several disadvantages compared to storing the length:

- The length of the string can not be calculated very fast. To calculate the length of the string, the whole string has to be scanned. This also causes problems when concatenating two strings. Because if we first calculate the lengths of the strings, the concatenation will be slower. But if we don't, we don't know in advance how much memory has to be allocated.

- A string can not be copied as fast, which makes the garbage collector slower. If we use a special end character, we can only copy a string character by character. If we know the length of the string, we can often copy many characters at once. If for example 4 characters are stored in one machine word, we can copy 4 characters at once.

- The string may not contain this special end character.

The advantages of using an end character are:

- A few bytes less memory is used.

- The last part of the string may be shared. If a new string has to be build which is the last part of a string, the string does not have to be copied, but the last characters of the string and the end character can be shared. For the ABC machine this is usually not useful. Although the ABC instruction `sliceS` could be executed faster if the slice is the last part of the string.

Because the only important advantage of using an end character is that it uses a few bytes less memory, we will not use this representation, but use the representation which includes the length of the string.

Consequently, a string consists of:
- a word which contains the length of the string. (in characters)
- an array of the characters of the string.

This string representation has also been used for the ABC code generator for the Sun [Weijers 1990].

Strings are allocated in the heap. A string node consists of a normal fixed size part, except that the pointer to the variable size part points to the string.

3.5. Representation on the MC68020.

In this section I will explain how the representations for register machines described above are represented on the MC68020. And also how these representation can be improved by using MC68020 specific properties, like efficient address arithmetic and efficient access to both words and long words.

The MC68020 has two sorts of registers: address registers and data registers. Only address registers can be used to access memory (see section 2.3.3), therefore the stackpointers and heap pointer should be allocated an address register. To implement the ABC instructions `jsr`, `jsr_eval` and `rtn` efficiently, the C-stackpointer should be allocated register A7, because the MC68020 `JSR`, `BSR` and `RTS` instructions use A7 as stackpointer to push or pop the return address. For the A- and B-stackpointers and heap pointer any free address register may be chosen.

The C-stack grows in the direction of low memory addresses, because the MC68020 system stack (A7) grows in this direction. For the A-stack and B-stack both directions can be chosen just as well. But by letting one stack start high in memory and grow in the direction of low memory addresses, and letting the other stack start low in memory and grow in the direction of high memory addresses, one of the stacks can become larger if the other one is small. Also a stack check can be implemented more efficiently, by testing that the stackpointer which started high in memory is still higher than or the same as the other stackpointer. In this implementation the A-stack starts high in memory and grows in the direction of low memory addresses, and the B-stack starts low in memory and grows in the direction of high memory addresses, but this could just as well have been the other way around.

The number of free long words in the heap can better be stored in a data register, because to allocate memory in the heap, we subtract the number of long words we want to allocate from this register, and then test if the garbage collector should be called by testing if the result became negative (see section 6.9). If we would use an address register for the number of free long words, we would need an extra instruction to test if the result became negative, because the MC68020 doesn't set the condition codes for a subtract from an address register. Another reason to use a data register is that, because the stackpointers and heap pointer are stored in address registers, there are not so many address registers left, but all data registers are still free.

Because this implementation is for the macintosh, register A5 can not be used, because it is used by the Macintosh to access the jump table and data area. On other machines this is not necessary, but then it may still be useful to let an address register point to the data area, because then values in the data area can be accessed faster using the d16(An) address mode instead of using the absolute address, because a displacement is only 16-bits and an address 32-bits. In that case the data area should not be larger than 64K. On the macintosh the data area can not be larger than 32K, but this is not a problem, because the data area is only used for descriptor elements, symbol records and constant strings, not for the heap and stacks.

In this implementation the following registers were used:

A3 A-stackpointer.
A4 B-stackpointer.
A5 data pointer, points to global data area.
A6 heap pointer, pointers to the top of the heap.
A7 C stackpointer.
D7 number of free long words in the heap.

The representation of nodes for register machines described in section 3.3 can be improved for the MC68020. This is so because descriptor elements are located in the data area, so that the descriptor address can also be represented by a 16-bit offset from the start of the data area instead of a 32-bit address. This would reduce the size of the representation of a descriptor address from 32 bits to 16 bits.
Using this representation pattern matching can be done faster. Because if we now have to compare a descriptor address, we can compare words (using `CMPI.W`) instead of comparing long words (using `CMPI.L`), which is faster (see 2.3.4 and 2.3.7). But using this representation would slow down access to the descriptor element. Because after loading the offset of the descriptor address from a node, the start of the data area has to be added to this offset to obtain the descriptor address (the start of the data area is always stored in register A5 on the Macintosh) Because the descriptor addresses are mainly used for pattern matching, and access to the descriptor element is not required very often, the representation using an offset is better. The offset to the descriptor address (address of the descriptor element) will be called the *descriptor*.

Consequently, a node is represented in the heap by: (in this order)

- 4 bytes: The address of the reduction code.
- 2 bytes: Not used. (filled with -1) Necessary to maintain long word alignment.
- 2 bytes: The descriptor.
- 4 bytes: If it is an integer, character or boolean node: an integer, character or boolean.
  If it is a real or string node: a pointer to a real or string. The real or string is also stored in the heap.
  Otherwise : a pointer to the argument(s) of the node. The argument(s) are 4 byte pointers to nodes and are also stored in the heap. If there are no arguments, this value is undefined, but has to be allocated so that this node can be overwritten by a node with arguments.

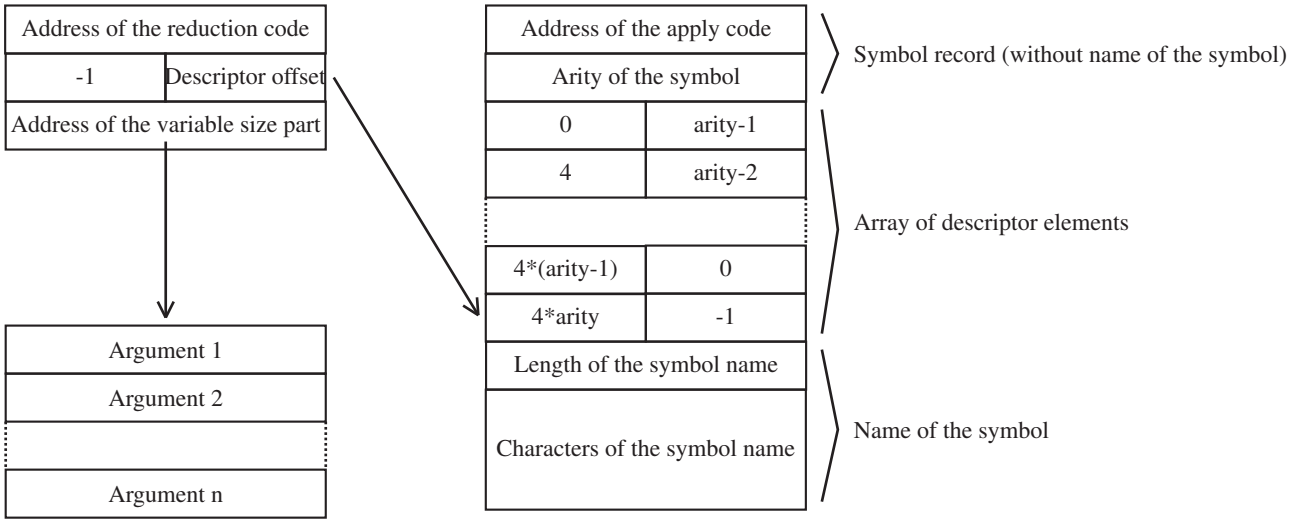
The representation of descriptor elements for register machines described in section 3.3 can also be improved for the MC68020. This is so because the MC68020 supports address arithmetic efficiently. Therefore the descriptor elements can be represented more compact. This can be done by storing the symbol record just before the array of descriptor elements. Then the end of the symbol record can be found using the number of arguments stored in the descriptor element, because the end of the symbol record = address of the descriptor element - (number of arguments stored in the descriptor element * size of a descriptor element). Now we no longer have to store the address of the symbol record in the descriptor element.
But even this representation can be improved by not storing the number of arguments in the descriptor element, but instead store the number of arguments * size of a descriptor element. Then the end address of the symbol record can be found faster (because the end of the symbol record is calculated by: address of the descriptor element - (number of arguments stored in the descriptor element * size of a descriptor element)).

During the evaluation of curried function applications it is necessary to compute: arity - 1 - number of arguments, therefore this number was also stored in the descriptor element. So now a descriptor element consists of the numbers: (array element number * size of a descriptor element) and (arity - 1 - array element number). Both are stored in a word.

Finally, the string representation of the name of the symbol was not stored in the symbol record, before the array of descriptor elements, but after this array. Otherwise a pointer to the string would have been necessary. Now the address of the string can be calculated by: descriptor address + first two bytes of descriptor element * 4 + second two bytes of descriptor element + 4.

Consequently, the following representation was chosen for the symbol record and the array of descriptor elements:

- Symbol record:
  - 4 bytes: address of the apply code of the symbol.
  - 4 bytes: arity of the symbol.
- Array [0 .. arity] of descriptor elements of:
  - 2 bytes: array element number (number of arguments of the node) * 4 (size of a descriptor element).
  - 2 bytes: arity - 1 - array element number (number of arguments of the node).
- String representation of the name of the symbol.



4. Run time system.

In this chapter the run time system is described. It explains how the garbage collector has been implemented, and what other things have been implemented in the run time system.

4.1. Garbage collection.

Because the ABC machine assumes there is an infinite heap, but a concrete machine only has a limited amount of memory, it is necessary to recycle memory. Memory which is no longer used should be reused again. Collecting the parts of the heap which are no longer used is called *garbage collection*.

Garbage collection will be done by a copying garbage collector, just as for the previous implementation. (on the SUN) [Weijers 1990]. The memory available for the heap is divided in two areas called *semispaces*. The nodes are stored in one semispace. When the semispace is filled with nodes, all nodes which are still needed for the execution of the program are copied to the other semispace, and this semispace becomes the current semispace.

The advantages of this garbage collection method are:
- the garbage collection is fast.
- no overhead during execution of the program, like updating reference counters.
- garbage collection automatically performs compaction.
- all nodes which are no longer needed can be recovered.

The disadvantages of this method are:
- only half the memory can be used.
- garbage collection can cause long pauses during the main computation.
- when the heap becomes fuller, the garbage collection has to be done more often and will be slower.

4.2. Remainder of the run time system.

The remainder of the run time system consists of:

- Code to allocate and free memory for the stack and the heap. Both the stacks, including the system stack, and the heap are long word aligned (see section 2.3.7).

- Code to perform output to a console. A simple console has been implemented, which consists of a window in which text is displayed. The output produced by the ABC instructions `print`, `print_sc`, `print_symbol`, `print_symbol_sc` and `dump` is displayed in this window. Input from a console and input from and output to files still has to be implemented.

- Code to compute the execution time and the time spend collecting garbage using a timer.

- Subroutines to implement string operations and conversions for the ABC instructions: `catS`, `sliceS`, `updateS`, `cmpS`, `eqS_a` and `fillS_symbol`.

- A subroutine to perform the reduction of AP nodes.
  In the previous implementation (for the Sun) [Weijers 1990] this was implemented using the ABC instructions `get_node_arity`, `get_desc_arity`, `push_ap_entry` and `add_args`. All these instructions have to access the descriptor element, and two of these instructions have to access the symbol record. But to perform the reduction of an AP node, all these accesses have to be done for the same node, so the address of the descriptor element was computed four times and the address of the symbol record was computed twice. The code generator can not detect this, because common

subexpression elimination (see section 5.1.6) has not (yet ?) been implemented, so some addresses were calculated more than once. By writing this code in assembly, these recalculations were eliminated, and the code for the reduction of AP nodes became faster.

- Subroutines to implement the ABC instructions: `eq_symbol`, `randomI`, `entierR` and `halt`.

The code to perform input and output to the console is written in C. All other code, including the garbage collector, has been written in MC68020 assembly language.

5. Possible code optimizations.

In this chapter possible code optimizations for the implementation of the ABC machine on the MC68020 are described. This is done by describing how the code generated by a very simple code generator can be optimized. With this very simple code generator I mean a code generator which translates every ABC machine instruction one at a time to a fixed sequence of machine code instructions (a kind of macro expansion). The code generated in this way is not very efficient, but we can use this code to examine how the code can be improved.

First general optimizations of ABC code are discussed. These are optimizations which can be performed for nearly all machines to which we want to translate ABC code. Then optimizations for register machines are discussed and finally MC68020 specific optimizations are discussed.

For most optimizations short examples are given. In these examples the following registers are given the following names:

A-stackpointer (A3)	ASP
B-stackpointer (A4)	BSP
C-stackpointer (A7)	CSP
Heap pointer (A6)	HP
Free long words counter (D7)	FC

5.1. General code optimizations.

Optimizations which can be performed for implementations of the ABC machine on nearly all machines are:

5.1.1. Optimizing the creation of nodes.

The ABC instruction `create` creates an empty node, by allocating space for it on the heap and initializing the allocated node. This initialization is necessary so that the garbage collector can determine that it is an empty node.
But if an empty node is always filled by an ABC `fill` instruction before a garbage collection can occur, initializing the node during a create is not necessary.

For example, MC68020 code for first creating and initializing, and then filling an integer node with value 0:

```
* allocate 3 long words in the heap:
      SUBQ.L    #3,FC                    ; subtract 3 from the number of free long
                                           words
      BMI       garbage_collect_1        ; if heap full, collect garbage
* create and initialize the node with empty:
      MOVEA.L   HP,A0                    ; move start address of new empty node to
                                           A0
      LEA       reduce_empty,A1          ; load empty node reduction address
      MOVE.L    A1,(HP)+                 ; store reduction address, reserve long
                                           word
      MOVE.L    #empty_descriptor*4,(HP)+ ; store descriptor * 4, reserve long word
      ADDQ.L    #4,HP                    ; reserve long word in the heap
* fill the node:
      LEA       _hnf,A1                  ; load reduction address of integer node
                                           (_hnf)
      MOVE.L    A1,(A0)+                 ; store reduction address in node
      MOVE.L    #integer_descriptor*4,(A0)+ ; store descriptor * 4 in node
      CLR.L     (A0)                     ; store integer value (0) in node
```

After the optimization, so without initializing:

```
* allocate 3 long words in the heap:
      SUBQ.L    #3,FC                     ; subtract 3 from the number of free long
                                            words
      BMI       garbage_collect_1         ; if heap full, collect garbage
* create a node:
      MOVEA.L   HP,A0                     ; move start address of new empty node to
                                            A0
      ADDA      #12,HP                    ; reserve 12 bytes for empty node in the
                                            heap
* fill the node:
      LEA       _hnf,A1                   ; load reduction address of integer node
                                            (_hnf)
      MOVE.L    A1,(A0)+                  ; store reduction address in node
      MOVE.L    #integer_descriptor*4,(A0)+ ; store descriptor * 4 in node
      CLR.L     (A0)                      ; store integer value (0) in node
```

For this example the optimized code is about 40 percent faster (estimation calculated using the execution times in section 2.3.7 or appendix B).

Very often even this can be improved by postponing the creation of the node to the `fill` instruction, so that creating and filling the node can be done at once, which is faster than first creating a node and later filling it.

For the same example as above, create and fill at the same time:

```
* allocate 3 long words in the heap:
      SUBQ.L    #3,FC                     ; subtract 3 from the number of free long
                                            words
      BMI       garbage_collect_1         ; if heap full, collect garbage
* create and fill the node:
      LEA       _hnf,A1                   ; load reduction address of integer node
                                            (_hnf)
      MOVE.L    A1,(HP)+                  ; store reduction address in heap, reserve
                                            long word
      MOVE.L    #integer_descriptor*4,(HP)+ ; store descriptor * 4 in heap, reserve
                                            long word
      CLR.L     (HP)+                     ; store integer value (0) in heap, reserve
                                            long word
```

For this example the optimized code is about 20 percent faster than without creating and filling at once, and about 65 percent faster than the first example with creating, initializing and filling (estimations calculated using the execution times in appendix B).

It is not always possible to fill and create a node at once if there is a reference to the address of the created node before the node is filled. In such a case the initialization after the creation is usually still not necessary. Sometimes the order of the instructions can be changed so that it is still possible to fill and create at once, but this is not possible for all nodes if there are cycles in the graph. For the following example this is not possible, because there is a cycle in the graph:

```
      create                                || create node 1
      create                                || create node 2
      push_a          1                     || copy node-id of node 1
      fill            descriptor_1 1 entry_1 1
                                            || fill node 2 with node 1 as argument
      push_a          0                     || copy node-id of node 2
      fill            descriptor_2 1 entry_2 2
                                            || fill node 1 with node 2 as argument
```

In this example one of the creates can not be postponed to the corresponding fill, because to create one of the nodes it is necessary to know the address of the other node.

If a node has to be created space has to be allocated for it on the heap by decrementing the number of free cells in the heap and, if there are not enough free cells left, calling the garbage collector (see section 6.9). If more than one node has to be created, this can be done faster by allocating space on the heap for all the nodes at once.

5.1.2. Jump optimization.

The Clean compiler often generates instruction sequences like:

```
      jmp_false      label1
      jump           label2
label1:
```

This can be replaced by:

```
      jmp_true       label2
label1:
```

If false was on the stack before the first jump instruction, the optimized code is usually not much faster, but if true was on the stack, only one jump instruction has to be executed instead of 2 jump instructions, so this is about twice as fast.

5.1.3. Strength reduction.

Integer multiplications by constants which are powers of 2 can be optimized by replacing them by shift instructions, because shift instructions usually are much faster. On the MC68020 a shift (`LSL`) instruction is about 12 times as fast as a multiplication (`MULS.L`) instruction.
Integer multiplications by small constants can be optimized by using shifts and additions instead, for example multiplying the value in machine register `D0` by 10 could be done with:

```
      MOVE.L    D0,D1                   ; copy the value to D1
      LSL.L     #2,D1                   ; multiply D1 with 4, now D1 = 4 * D0
      ADD.L     D1,D0                   ; add D1 to D0, now D0 has been multiplied
                                          by 5
      ADD.L     D0,D0                   ; multiply D0 by 2, now D0 has been
                                          multiplied by 10
```

This is often faster than using the multiply instruction, but often also makes programs larger. On the MC68020 this code is 1/3 longer, but about 4 times as fast. (Estimated using the execution times in appendix B)

Integer divisions by constants which are powers of 2 can also be done faster by shifting. For positive values a division by 2 ** n can be done with an arithmetic shift right by n bits. But for negative values we first have to add 2 ** n - 1 before shifting, due to differences in rounding. For example to divide `D0` by four:

```
      TST.L     D0                      ; test D0
      BPL.S     label1                  ; branch if D0 is negative
      ADDQ.L    #4-1,D0                 ; add 4-1 to D0 (if D0 is negative)
label1:
      ASR.L     #2,D0                   ; shift right two bits
```

Again this is often faster, but often also makes programs larger. For the MC68020 the example is about 6 times as fast, but the code is also 1/3 longer.

Floating point multiplication and division by constants which are powers of 2 can sometimes be optimized. On the MC68881 floating point coprocessor for example the `FSCALE` instruction, which performs a multiplication by 2 ** n, can be used to optimize these multiplications and divisions. But for the MC68881

floating point multiplications by small constants, which are not powers of 2, can not be optimized by replacing the floating point multiplications by additions and shifts (FSCALES). This is because on the MC68881 additions and shifts on floating point values are not so much faster than multiplications as they are on integers on the MC68020.

A floating point division by a constant c can be optimized to a multiplication by the constant (1/c), because usually a multiplication is executed faster than a division. (of course 1/c has to be computed during code generation) But the result may be slightly different due to rounding errors.

5.1.4. Constant folding.

Computations on constants can sometimes be done compile-time instead of run-time. For example:

```
pushI           2
pushI           3
mulI
```

could be replaced by:

```
pushI           6
```

5.1.5. Other algebraic optimizations.

(strength reduction and constant folding are also algebraic optimizations)
Sometimes computations can be optimized by using algebraic transformations.

For example:	Can be replaced by:
x+0, 0+x, x-0, x*1, 1*x and x/1	x
x*0 and 0*x	0
0-x	-x
a-a	0
a*b+a*c	a*(b+c)

(a*b+a*c and a*(b+c) are not always the same when overflows or underflows occur)

5.1.6. Common subexpression elimination.

Sometimes the same calculation is done more than once, for example in (a*b)+(a*b) the multiplication of a by b is done twice. In such a case it is often faster to do the calculation only once and to remember the result, for example: (c=a*b)+c. Because the Clean compiler does not eliminate such recalculations (usually called common subexpressions), these recalculations can also occur in ABC code.

Another example is that when a node is created or filled, the address of the code, which evaluates the node to head normal form, has to be stored in the node. If more nodes are created which have the same evaluation address, the code can often be made faster, for example on a register machine by first loading the address in a register.

5.1.7. Removal of unused code.

Sometimes code is generated which will never be executed, removal of this code does not make a program faster, but does make it smaller. For example the node entry of the code generated for a Clean function is sometimes not used. Then the code which is only used in case the function is called using the node entry can be removed.

5.1.8. In-line code substitution for small functions.

In-line code substitution is replacing a call to a function by the body (code) of the function. This is a very useful optimization for small functions. Because then no instructions have to be executed to call the function, return from the function and pass the parameters and results. And for register machines the code often also becomes better because then no registers have to be saved before the function call and loaded again after the function call, which is often necessary if a function is called.
For small functions the program doesn't become much larger, because only small pieces of code are substituted. The program could even become smaller, because the instructions which pass the parameters and call the function (and save and load registers for a register machine) are removed.
In-line substitution for large functions would make the program too large, and the speed improvement would only be small. But if a large function is called only once (and is not recursive), in-line code substitution could be used. Because after this substitution the code of the function is no longer used, and can be removed.
When doing in-line code substitution we should be careful with recursive functions, because the code generator will enter an infinite loop if recursive functions are substituted over and over again.

In-line code substitutions can probably better be done by the Clean compiler than by the code generator. The Clean compiler already does code substitutions for delta-rules, which are the most important ones. But doing more in-line code substitutions will often improve the efficiency of the code considerably. For example if Clean functions are used which compute a constant, for example: `NumberOfElements -> 10;` .

5.1.9. Removing pointers from the A-stack.

If a pointer to a node is on the A-stack and the pointer will never be used any more, the pointer could be removed from the A-stack by reordering the values on the A-stack or by overwriting it with a nil value, for which the garbage collector should now test.

Because now the garbage collector will remove more nodes from the heap, the garbage collector will be called less frequently. Unfortunately the reordering or overwriting slows down the execution of the program. Normally this will probably be more expensive than the gain in time spent garbage collecting, but if the reordering can be done without cost or the pointer points to a large graph, it may be worthwhile.

For a register machine this optimization can sometimes improve the code. If a node-id on the A-stack is stored in a register, and a function is called, it is often necessary to store the node-id in the register in memory. But if this node-id will never be used any more, this store is not necessary. If we now can remove this node-id from the A-stack without cost, by reordering the A-stack, the code will be faster, because the store can be removed.

5.2. Possible code optimizations for register machines.

Optimizations which can be performed for implementations of the ABC machine on register machines are:

5.2.1. Better use of registers.

On a register machine it is more advantageous to store values in registers instead of in memory, because:

- A register machine can address values in registers a lot faster than values in memory. This also applies to the MC68020 (see section 2.3.7).

- On values in registers all computations can usually be done directly, but if a value is in memory, it is often necessary first to move the value into a register before the computation can be done. On the MC68020 it is for example not possible to add two values in memory with an `ADD` instruction, but it is possible to add two values in registers with an `ADD` instruction.

- Values in memory can usually not be used to address memory (used as a pointer) efficiently. On the MC68020 this is only possible using the (slow) memory indirect addressing modes (e.g. `([10,An],20)`), but even in this case it is faster first to move the value (pointer) in memory to an address register and then use an address register indirect addressing mode (see section 2.3.7). But values in registers can usually be used to address memory. On the MC68020 address registers can be used to address memory, even using a displacement, postincrement or predecrement is possible (see section 2.3.3).

Therefore the quality of the generated code is determined for a great deal by how well use is made of registers.

However, the code generated by a very simple code generator would only uses registers for the stackpointers, the heap pointer and to hold values during the execution of one ABC instruction. Registers are not used to pass the result from one ABC instruction to another ABC instruction.

But it is often possible to improve the code by keeping a value in a register, instead of pushing it on the stack, for example the ABC code:

```
addI
addI
```

would on the MC68020 result in:

```
MOVE.L     -(BSP),D0
ADD.L      D0,-4(BSP)
MOVE.L     -(BSP),D0
ADD.L      D0,-4(BSP)
```

using a very simple code generator. If we would pass the result of the first addI to the second addI in a register, the (more efficient) code would be:

```
MOVE.L     -(BSP),D0
ADD.L      -(BSP),D0
ADD.L      D0,-4(BSP)
```

The optimized code of the example is about 40 percent faster on the MC68020 (estimated using the execution times in appendix B).

5.2.2. Passing parameters and results of functions in registers.

When a function is called, jumped to or left, using a `jsr`, `jsr_eval`, `jmp`, `jmp_eval`, or `rtn` instruction, the parameters for and/or result of this function have to be passed. The code generated by the Clean compiler passes the parameters and results on top of the A-stack and B-stack. Because a function may be called from several places in a program, even from an other module, a fixed parameter passing convention is used. The simplest solution is to store all stack elements in memory on the stacks when a function is

entered or left, and so to pass the parameters and results in memory on the stacks. But passing parameters and results in registers is usually more efficient, because:

- Computing the parameters and results is usually more efficient, because computing a value in a register can be done faster than computing a value in memory.
- Parameters and results can be accessed faster, because registers can be accessed faster than memory locations.
- Parameters and results can often be popped from the stack without having to adjust stackpointers, which is usually necessary when parameters and results are passed in memory.

5.2.3. Eliminating unnecessary copies.

Sometimes the Clean compiler generates code which copies values on a stack, because the values are not on top of the stack and an instructions expects its argument(s) on top of the stack. Sometimes values are moved on a stack which are never used. Some examples are:

- Filling nodes:

```
push_a          1
push_a          3
fill            descriptor 2 entry 5
```

  The two values which are pushed on the A-stack don't have to be copied first, but could be stored in the node by the fill instruction by copying from the original location of the two values on the A-stack.

- Store the value in the right stack location immediately after computation of the value:

```
addI
update_b        0 3
pop_b           1
```

  The value calculated by `addI` should not be stored on top of the B-stack and later moved to location 2, but immediately at location 2 of the B-stack. (location 3 becomes location 2 after popping one element off the stack)

- Pushing all the arguments of a node, but not using all of them:

```
push_args       0 2 2
pop_a           1
```

  Only the second argument should be pushed on the A-stack.

5.2.4. Optimizing booleans.

If a conditional jump has to be taken depending on a boolean which is the result of a comparison, it is not necessary to calculate a boolean value, but the conditional jump can use the condition codes of the register machine. For example for:

```
eqI_b           +1000 0
jmp_false       label1
```

the following code would be generated for the MC68020: (if the top of the stack is in register D0, and the boolean is computed in register D1, and a boolean is represented by -1 for true and 0 for false)

```
CMPI.L     #1000,D0
SEQ        D1
EXTB.L     D1
TST.L      D1
BEQ        label1
```

but if the condition codes are used the following code is generated:

```
CMPI.L     #1000,D0
BNE        label1
```

On the MC68020 a conditional jump without creating a boolean is about 1.8 times as fast (estimated using the execution times in appendix B).

The ABC machine doesn't have instructions to test for 'not equal', 'less than or equal' and for 'greater than or equal'. These instructions are implemented by a test for 'equal', 'greater than' or 'less than' and a notB instruction. But a register machine usually has instructions to test for 'not equal', 'less than or equal' and for 'greater than or equal', so these tests can be optimized so that the not is not necessary. This optimization is possible on the MC68020.

5.2.5. Changing the evaluation order.

Because a register machine only has a limited number of registers, not all values can be stored in registers. Sometimes we will want to store a value in a register, but will not be able to do so, because all registers have been used. In such a case it is sometimes possible to change the order of the instructions so that it is possible to store the value in a register after all.

Another improvement we may obtain by changing the order of instructions, is the elimination of some move instructions. This could happen if a computation is done using values which will also be used later. Then often a move is necessary to copy one of the values which are used by the computation, because doing a computation often overwrites one of its arguments. For example an `ADD` instruction changes its second argument, so that if the value (before the addition) of the second argument has to be used later, it has to be copied before the addition is done. If the order of the instructions is changed so that a (later) use of a value is moved before the computation, this move can often be removed.

I will now give an example which illustrates both improvements. Assume we want to compute `D0+D1` and `D0+D2` in any register, and `D1` is used after these computations, but `D0` and `D2` are not. We could first compute `D0+D1` and then `D0+D2` using:

```
MOVE.L     D0,D3                    ; copy D0 to D3
ADD.L      D1,D3                    ; add D1 to D3, D3 now contains D0+D1
ADD.L      D2,D0                    ; add D2 to D0, D0 now contains (old)D0+D2
```

But it would be better first to compute `D0+D2` and then `D0+D1`:

```
ADD.L      D0,D2                    ; add D0 to D2, D2 now contains D0+(old)D2
ADD.L      D1,D0                    ; add D1 to D0, D0 now contains (old)D0+D1
```

In the latter case we have used one register less (`D3` is not used) and we have used one move instruction less.

Consequently, the order of the instructions (the evaluation order) is important to generate efficient code.

5.2.6. Optimizing jsr_eval instructions.

The ABC instruction `jsr_eval` evaluates a node to head normal form by calling the code at the reduction address (reduction code) of the node. (see section 2.2.6) But often `jsr_eval` is used to evaluate a node which is already in head normal form. (although the Clean compiler tries to prevent this) In such a case, nothing has to be done to evaluate the node, and so the reduction code consists of just a return instruction.

If this happens we unnecessarily perform a subroutine call (`JSR` for the MC68020) and a return from subroutine instruction (`RTS` for the MC68020). But more importantly, for register machines often registers have to be saved before the reduction code is called and loaded again after the call.

To prevent this, if we first test if the reduction address of the node is the reduction address of the reduction code of a node in head normal form. And only save registers, call the reduction code and load registers if this is not so.

So if the node is already in head normal form, we only have to test the reduction address of the node, instead of saving registers, calling a subroutine, return from the subroutine and loading the registers again. For most register machines, just testing the reduction address can be performed much faster.

If the node is not yet in head normal form, we only have to perform an extra test for the reduction address. So in this case the code becomes only a little bit slower.

Because for many programs often nodes are 'evaluated' using `jsr_eval` which are already in head normal form, programs will often be executed faster. (sometimes 1.5 times as fast) Other programs will only be executed slightly slower.

For many machines we can perform the test whether the reduction address is the reduction address of a node in head normal form faster if we represent the reduction address of a node in head normal form by 0. For example on the MC68020, we could then test for such a reduction address by loading the reduction address in a data register (with `MOVE.L`), and then using a branch on equal (`BEQ`) instruction. Instead of using a compare (`CMP.L`) instruction and a branch on equal instruction, which is slower. Then also filling a node in head normal form can often be done faster, because storing zero in memory can usually be done faster (for the MC68020 using a `CLR.L` instruction) than storing an address in memory. If we use this representation, the `jmp_eval` instruction also has to test for a zero reduction address. This makes `jmp_eval` slower, but this instruction is used seldom.

This optimization has been implemented, including representing the reduction address of a node in head normal form by 0.

5.3. MC68020 specific code optimizations.

Optimizations which can be performed for implementations of the ABC machine on the MC68020 are: (most of these optimizations have already been described in section 2.3.7)

- Using the postincrement ((An)+) and predecrement (-(An)) addressing modes for stack accesses, filling and creating nodes and loading the arguments of a node.

- Using the `MOVEQ`, `ADDQ` and `SUBQ` instructions when possible, instead of the MOVE, ADD and SUB instructions.

- Using the `MOVEM` instruction when many registers have to be moved to or from consecutive memory locations.

- Using:
```
MOVEQ      #100,D1
CMP.L      D1,D0
```
instead of:
```
CMPI.L     #100,D0
```
to compare a long word to a constant between -128 and 127 (except for zero, use a `TST` instruction).

- Using `CLR <ea>` instead of `MOVE #0,<ea>` and using `TST <ea>` instead of `CMPI #0,<ea>`.

- Trying to use `CMP <ea>,Dn` instead of `CMPA <ea>,An` to compare words and long words.

- Using `BRA` instead of `JMP (d16,PC)`, and using `BSR` instead of `JSR (d16,PC)`.

- Using `LSL` instead of `ASL` when possible.

6. Generating code.

In this chapter is described how the code generator generates code. Briefly, this is done in the following way.

First the ABC instructions are divided into blocks, called *basic blocks*. And a directed acyclic graph is constructed for such a basic block. Such a graph represents the computations performed by the basic block. This is done so that the evaluation order can be determined. Why and how the ABC instructions are divided into basic blocks, and why a graph is used, is explained in section 6.1.

Then the order in which such a graph will be evaluated is determined using an adapted *labeling algorithm*. How this is done is explained in section 6.2.

Then in section 6.3 is explained how the ABC instructions are represented in the graph.

After that, in section 6.4, is explained how code *intermediate code* is generated from this graph. This intermediate code is very close to MC68020 machine code. But in this intermediate code an unlimited number of address and data registers may be used. How the code generator determines whether to use a data register or an address register of the MC68020 if a register is used is described. Also is explained how the following optimizations are performed: optimizations of the creation of nodes by `create` instructions, optimizing the use of booleans by using condition codes, optimizing the use of small constants and how generating unnecessary store instructions is prevented.

Before the graph for a basic block is constructed, the code generator determines which values are stored in registers at the start of the basic block. And after the graph has been constructed, the code generator determines which values are stored in registers at the end of the basic block. This is called *global register allocation* and is explained in section 6.5. How parameters and results of functions are passed is also explained in this section.

Because the intermediate code may use an unlimited number of data and address registers, and the MC68020 only has 8 data and address registers, the *local register allocator* changes the intermediate code of a basic block so that no more than 8 address registers and data registers are used. It does this by changing the register numbers of the registers used by the intermediate instructions and by inserting instructions to load and store values in registers from/into memory. This is explained in section 6.6.

After local register allocation has been performed, accesses to the stacks are optimized by using the postincrement and predecrement addressing modes of the MC68020 by changing the intermediate code. This is described in section 6.7.

After that jumps are optimized by replacing a branch instruction followed by a jump instruction by one branch instruction if possible, which is described in section 6.8.

Then in section 6.9 is described how the garbage collector is called.

And finally, in section 6.10. is described how object code (for the linker) is generated from the intermediate code. While generating this object code many very simple MC68020 specific peephole optimizations are performed, which are also described.

6.1. Constructing a dag for every basic block.

In this section is explained how the order of the ABC instructions may be changed, so that the result of the code remains the same. To be able to do this, graphs are build representing the computations performed by a sequence of ABC instructions.
To build these graphs, first the ABC instructions are divided into blocks, called *basic blocks*. To construct these basic blocks, the ABC instruction are divided into two classes: *instructions without side effects* and

instructions with side effects. For every basic block a directed acyclic graph (*dag*) is constructed. Such a graph represents the computations performed by the basic block. To build this graph the A and B stack of the ABC machine are simulated by the code generator.

6.1.1. Conditions for changing the evaluation order.

In section 5.2.5 we already saw that the evaluation order is important to generate efficient code. But by changing the order of the instructions we may also change the meaning of the program. Of course, we do not want this to happen.

In order not to change the meaning of the program by changing the evaluation order, we can not move an instruction:
- Before an instruction of which the result may be used by the instruction to be moved.
- After an instruction which may use the result of the instruction to be moved.

These are the only requirements for instructions which:
- Only use the values of their arguments to compute a result.
- Store its result in a location which is never changed by an other instruction.
- Don't change or use anything else. For example, the instruction may not change a stackpointer or jump to an other part of the program.

6.1.2. Problems on a stack machine when changing the evaluation order.

Changing the evaluation order causes many problems on a stack machine like the ABC machine, because:

- It is difficult to move instructions, because:
  - Many instructions change a stackpointer as side effect. If we want to move such an instruction, we may have to adjust many instructions and add instructions to reorganize the stack, because the stack layout changes. For example:

```
pushI          1                    || push the integer 1
push_b         2                    || push (copy) the third element of the B-
                                       stack
push_b         2                    || push (copy) the third element of the B-
                                       stack
addI                                || pop two elements (integers) and then
                                       push the sum of these integers
addI                                || pop two elements (integers) and then
                                       push the sum of these integers
```

    If we want to move the `pushI 1` instruction before the last `addI`, we have to change the `push_b 2` instructions into `push_b 1` instructions:

```
push_b         1                    || push (copy) the second element of the
                                       B-stack
push_b         1                    || push (copy) the second element of the
                                       B-stack
addI                                || pop two elements (integers) and then
                                       push the sum of these integers
pushI          1                    || push the integer 1
addI                                || pop two elements (integers) and then
                                       push the sum of these integers
```

  - Many instructions expect some (or all) of their arguments on top of a stack. If we want to move such an instruction, we may have to insert instructions to move the argument(s) on top of a stack and reorganize the stack(s) after the instruction.

- It is difficult to determine where instructions may be moved. The result of most of the instructions only depends on the arguments. These instructions may be moved to a location in the code where the arguments have been computed and are on the stack, and where all instructions which use the result of this instruction are after this instruction. But it is difficult to determine these locations, because:

    - To find where an argument is computed on a stack machine requires a backward search from the instruction which uses the argument to the instruction which computes the argument. While searching we have to keep track of where the argument is on the stack, so we have to do a simple stack simulation.

    - To find where the result of an instruction is used on a stack machine requires a forward search from the instruction which computes the result to the instruction which uses this result. Again we have to do a simple stack simulation while searching.

6.1.3. Simulating the A- and B-stack.

A way to solve the problems described above is to give every element on the stack a name, i.e. treat each stack element as a variable, and replace the ABC instructions by instructions which don't use a stack, but which arguments are variables. Then changing the evaluation order becomes much easier.

The example in the previous section now becomes: (`n1`, `n2`, `n3`, `n4` and `n5` are the variables for stack locations 1, 0, -1, -2 and -3 at the beginning of the code)

```
n3 := 1                 pushI 1
n4 := n1                push_b 2
n5 := n2                push_b 2
n4 := n5 + n4           addI
n3 := n4 + n3           addI
```

We can now move the `n3 := 1` instruction before the last add instruction (`n3 := n4 + n3`) without having to adjust other instructions:

```
n4 := n1
n5 := n2
n4 := n5 + n4
n3 := 1
n3 := n4 + n3
```

The advantages are:
- We can now move instructions without having to change other instructions.
- We can determine easier where an instruction may be moved, because we can easily determine where an argument of an instruction is computed.

To construct this code using variables, the stacks have to be simulated during code generation. At the same time we have to assign variables to stack locations and remember for every stack location which variable has been assigned to it.

6.1.4. Local variables.

By using variables it is possible to move instructions without having to adjust other instructions, but unfortunately there still are unnecessary restrictions for the way in which we can reorder the instructions. For example: (`n1`, `n2`, `n3`, and `n4` are the variables for stack locations 2 ,1, 0 and -1 at the beginning of the code)

```
n4 := n1                push_b 2
n3 := n3 + n4           addI
n4 := n2                push_b 1
```

We can not move the `n3 := n3 + n4` instruction after the `n4 := n2` instruction, because the `n4 := n2` instruction changes the value of variable `n4`. If we would use a local variable `l1` to hold the result of the first instruction we can move the instruction after all. The example now becomes:

```
l1 := n1                push_b 2
n3 := n3 + l1           addI
n4 := n2                push_b 1
```

After moving the `n3 := n3 + l1` instruction:

```
l1 := n1
n4 := n2
n3 := n3 + l1
```

By introducing a new local variable for every result, we obtain more freedom for determining the evaluation order. But we can't simply replace all variables (what we called variables in section 6.1.3, from now on called global variables) by local variables, because at the end of the code the global variables should have the same value as they should have had without the introduction of the local variables. Therefore for every global variable which was changed by the code, we add an instruction at the end of the code. This instruction assigns a local variable, which contains the value which should be in the global variable at the end of the code, to the global variable. For this local variable we use the last local variable which was introduced by replacing this global variable.

Consequently, the example becomes: (before moving the instruction)

```
l1 := n1                push_b 2
l2 := n3 + l1           addI
l3 := n2                push_b 1
n3 := l2                ; extra assignment to n3
n4 := l3                ; extra assignment to n4
```

By introducing these extra assignment instructions, this code gets worse. But code is generated for the MC68020 from these instructions in such a fashion that the MC68020 code doesn't get worse, because most unnecessary assignments are eliminated.

6.1.5. Dags (directed acyclic graphs).

In section 6.1.1 we have seen two simple conditions for changing the order of instructions which meet the following requirements:

- Only use the values of their arguments to compute a result.
- Store its result in a location which is never changed by an other instruction.
- Don't change or use anything else.

We will now examine which ABC instructions meet these requirements. These ABC instructions will be called *instructions without side effects*. For some ABC instructions, which do not meet these requirements, we can use the same conditions for changing the evaluation order as for instructions without side effects.

These instructions are also called instructions without side effects. They are described in section 6.1.6. The other ABC instructions will be called *instructions with side effects*. They are described in section 6.1.7.

Most ABC instructions use only the values of their arguments and only change one element of a stack and eventually change some stackpointers by a constant. By introducing global and local variables as described in the previous sections, these instructions meet the requirements of instructions without side effects. Because the result is always stored in a local variable, and a new local variable is introduced for every result, the result is stored in a location which is never changed by an other instruction. The only other things these instructions may change are stackpointers, but by using variables this is not a problem. So these ABC instructions are instructions without side effects.

We will now assume all ABC instructions are like this. Then every instruction computes one result, and a new local variable is created every time a value is calculated, so there is a one to one correspondence between a local variable and the instruction which stores its result in that local variable. If we replace every reference to a local variable by a reference to this instruction, we can remove all local variables.

Then every one of these instructions can be seen as a node in a graph. If we also make nodes for every global variable and constant used as an argument, every argument would be a pointer to a node. Then all the nodes together constitute a *dag*, i.e. a directed acyclic graph. For every global variable which is changed by the code we now have a node in the dag, which represents the computation of the value which has to be stored in that variable. This is because the extra assignment at the end of the code for every changed global variable has now become a node. If we then label these nodes with the corresponding global variable, we obtain a dag in which for every global variable there is exactly one node labeled with this global variable.

This dag is as follows:
- The leaves of the dag are global variables and constants which are used during the computations.
- The other nodes (the interior nodes) in the dag consist of a function (like add and sub) and the arguments for this function. These arguments are pointers to nodes in the dag.
- A node is labeled by all global variables which should contain the value represented by the dag after the code has been executed.

6.1.6. Remaining ABC instructions without side effects.

At the end of the previous section we assumed that all ABC instructions only use the values of their arguments and only change one element of a stack and eventually change some stackpointers by a constant. For these instructions it is not difficult to determine where they may be moved to (see section 6.1.1), and they can be moved within the code and represented in the dag without problems. But for the other ABC instructions some of these things cause problems.

In this section I will describe the ABC for which we can easily solve these problems. These instructions can then be treated just like instructions without side effects, and therefore we will also call these instructions instructions without side effects. In the next section the ABC instructions for which we can not easily solve these problems are described.

The ABC instructions for which we can easily solve the problems are:
- `create`, `add_args` and `del_args`.
- all fill instructions and `set_entry`.
- `push_args` and `repl_args`.
- instructions which compute a real.

The problems for these instructions are described below.

Create, add_args and del_args.

These instructions do not only change a value on the A-stack, but also change the heap, because they create a new node in the heap. But this doesn't cause any problems, because these instructions don't change

anything that is already in the heap, but only add a new node to the heap. So we can ignore the changes in the heap. Then these instructions only change one element of the A-stack, because the node-id of the created node is pushed on the A-stack.

All fill instructions and set_entry.

These instructions change the heap, because they overwrite a node (or part of a node in the case of `set_entry`) in the heap, but don't change any element of a stack. This causes the following problems:

1. Because no element of a stack is changed, we can't connect a node which represents such an instruction to the dag, because there is no result on any stack. But this can easily be solved by considering the node-id of the node which is filled as the result of the instruction, although this node-id is not actually changed. This node-id is on the A-stack.

2. When changing the evaluation order we may in general not change the order of these instructions, for example we may not exchange the following two instructions:

```
fillI          1 0                  || fill (overwrite) the node of which the
                                       node-id is at position 0 of the A-stack
                                       by INT 1
fillI          2 1                  || fill (overwrite) the node of which the
                                       node-id is at position 1 of the A-stack
                                       by INT 2
```

   This reason for this is that if positions 0 and 1 of the A-stack contain the same node-id, the node corresponding to this node will have been overwritten by a INT 2 node after executing the code if the instruction have not been exchanged, but if the instructions have been exchanged the node will have been overwritten by an INT 1 node. So exchanging instructions may change the result of the code, and is therefore not allowed.

   Fortunately the Clean compiler generates code in such a fashion that the situation described above will never occur. Because we only have to generate code for ABC programs generated by the Clean compiler, this is not a real problem.

3. Another problem is that if one of these instructions is represented in the graph by a node, this node can be connected as described above (1.), but there may still not be a path from a node labeled by a global variable to the node representing this fill or `set_entry` instruction. If we would generate code only for the nodes of the dag, to which there is a path from a node labeled by a global variable, to prevent generating code for unnecessary computations, no code would be generated for this fill or `set_entry` instruction. For example:

```
create                              || create a node and push its node-id on
                                       the A-stack
create                              || create a node and push its node-id on
                                       the A-stack
push_a         0                    || push (copy) the node-id on top of the
                                       A-stack
fill           descriptor 1 label 2 || fill (overwrite) the node created first
                                       with the descriptor, label and only
                                       argument the node-id of the node
                                       created second and pop this node-id
                                       from the A-stack
fillI          1 0                  || overwrite the node created second by an
                                       INT 1 node.
pop_a          1                    || pop the node-id of the node created
                                       second from the A-stack.
```

   There is no path to the `fillI 1 0` instruction from a node which is labeled by a global variable because the node-id is popped from the A-stack by the last instruction. If we would generate code only for the nodes of the dag, to which there is a path from a node labeled by a global variable, to

prevent generating code for unnecessary computations, no code would be generated for this fill instruction. But code should have been generated for this instruction, because the node which is to be filled is connected to the graph which will be constructed run time by the code, because it is the argument of the other node which is created by this code.

Fortunately the Clean compiler generates the `create` and fill instructions in such an order, that the situation described above will never occur. Because we only have to generate code for ABC programs generated by the Clean compiler, this is not a real problem.

Push_args and repl_args.

These instructions don't always compute one result, but often compute many results (the arguments). This makes representing these instructions by nodes in the graph more difficult, but not impossible. It doesn't cause any problems for changing the evaluation order.

Instructions which compute a real.

The problem with these instructions is that they compute two elements on the B-stack, because a real consists of two elements on the B-stack. This makes representing these instructions by nodes more difficult, but not impossible.

6.1.7. ABC instructions with side effects.

In this section the ABC instructions are described for which we can not easily solve the problems. These ABC instructions will be called instructions with side effects. They are:

- `catS`, `sliceS` and `updateS`.
- `jmp`, `jmp_eval`, `jmp_false`, `jmp_true`, `rtn` and `halt`.
- `jsr`, `jsr_eval`, and `ccall`.
- `push_args_b` and `repl_args_b`.
- `print`, `print_sc`, `print_symbol` `print_symbol_sc` and `dump`.
- `fopen`, `fclose`, `fgetC`, `fgetS` and `fputC`.

The problems for these instructions are described below.

CatS, sliceS and updateS.

These instructions change the heap, because they overwrite a node in the heap, but don't change any element of a stack. This causes the same problems as for the fill instructions and `set_entry`, but these problems can be solved (see section 6.1.6).
But these instructions can only be implemented by many MC68020 instructions. Consequently these instructions have to be implemented by using a subroutine. Representing subroutine calls in the graph causes many problems and often does not allow us to generate better code.

Jmp, jmp_eval, jmp_false, jmp_true, rtn, and halt.

These instruction change (or may change for `jmp_false` and `jmp_true`) the flow of control. Therefore it is usually not possible to move instructions from before such an instruction after that instruction, or from after such an instruction before that instruction. For example in general we can't exchange the instructions: `n1 :=1` and `jmp label`. In some rare cases an instruction could be moved, for example the `n := 1` instruction

may be moved after the `label` if no other instructions jump to `label`, but it is very difficult to determine when this is possible and will seldom improve the code.

Jsr, jsr_eval and ccall.

These instructions call a subroutine. Because in general we don't know what the subroutine does, we can't move instructions from before such a call instruction after that instruction, or from after such a call instruction before that instruction. And we also can't change the order of the call instructions.

Push_args_b and repl_args_b.

These instructions don't change the A-stackpointer by a constant, but by a value which depends on a value on the B-stack, so the change of the A-stackpointer is not known compile time. Therefore it is usually not possible to move instructions, which access a value on the A-stack, from before a `push_args_b` or `repl_args_b` instruction after that his instruction, or from after a `push_args_b` or `repl_args_b` instruction before that instruction. For example we can't exchange a `n1 := 1` (with `n1` on the A-stack) and a `push_args_b` instruction, because after the `push_args_b` we can no longer determine where `n1` is on the A-stack, so we don't know where to store the `1`.

Print, print_sc, print_symbol, print_symbol_sc and dump.

These print instructions don't compute a result on a stack. It is not possible to exchange print instructions, because the output would not always be the same. But other instructions could be moved from before a print instruction after that instruction, or from after a print instruction before that instruction. The `dump` instruction is an exception, because it also changes the flow of control, because it stops the execution of the program. And these instructions have to be implemented by calling a subroutine.

Fopen, fclose, fgetC, fgetS and fputC.

These instructions may cause problems when the order of these instructions is changed, for example because some of them change the position where characters are written to and/or read from in a file. They also have to be implemented by calling a subroutine.

6.1.8. Division into basic blocks.

In the previous sections we divided the ABC instructions into instructions without side effects and instructions with side effects. For the instructions without side effects the possible evaluation orders can easily be determined using the two conditions in section 6.1.1 and these instructions can be represented in the dag. For the instructions with side effects this is not possible.

Consequently we can rearrange the order of a sequence of instructions without side effects, but we can't move an instruction from before an instruction with side effects after that instruction, or from after an instruction with side effects before that instruction. So to determine the evaluation order for a code sequence we can determine the evaluation order for every sequence of instructions without side effects, which is as long as possible and which is part of the whole code sequence, separately.

Labels also restrict the possible evaluation orders, because in general we can't move an instruction from before a label after that label, or from after a label before that label. So to determine the evaluation order of an instruction sequence which contains labels, we can determine separately the evaluation order of the code

sequences from the beginning of the code sequence to the first label, from the first label to the second, ... , from the second last label to the last label and from the last label to the end of the code sequence.

Consequently, to determine the evaluation order of a code sequence, we can divide the ABC code sequence into *basic blocks* (basic blocks are also used by [Aho et al. 1986], but these are not exactly the same). These basic blocks consist of a sequence of instructions of which only the first instruction may be labeled (with any number of labels) and of which only the last instruction may be an instruction with side effects, all other instructions should be instructions without side effects. These basic blocks should be made as long as possible. To determine an evaluation order for the whole code sequence, we only have to determine an evaluation order for every basic block separately. If the last instruction of the block is an instruction with side effects, this will of course always remain the last instruction of the block.

Note that such a basic block can only be executed by starting with the first instruction of the block and can only be left by the last instruction of the block, because a basic block can only have labels before the first instruction and instructions which change the flow of control can only be the last instruction of a basic block, because these are instructions with side effects.

6.1.9. Constructing the dag.

To construct the dag from a basic block we can use an adapted version of algorithm 9.2 of [Aho et al. 1986]. The original algorithm also catches common subexpressions, but our algorithm does not.

We assume all instructions are of the form: `x = f (a1,a2,..,an)`, where `f` is a function with arguments a_1, a_2, .., a_n. This instruction computes `f` with arguments a_1, a_2, .., a_n and assigns the result to `x`. A_1, a_2, .., a_n and `x` are variables.

Initially the dag is empty and the function `node`, which argument is a variable and returns a node, is undefined for all variables. To construct the dag: do for every instruction in the basic block, starting with the first instruction, the following:

1. Do for every argument `ai` in { a_1, a_2, .., a_n }:
   If `node (ai)` is undefined, then create a node with variable a_i and let `node (ai)` be this node.
2. Construct a function `f` node with arguments (`node (a1)`, `node (a2)`, .., `node (an)`).
3. Let `node (x)` be this constructed node.

Finally do for all variables:
if `node (variable)` is not undefined then label `node (variable)` with this label.

To describe the dag we have build, I use the notion *cdag*. With a cdag (connected dag) I mean a subgraph of a dag which consists of all the nodes to which there is a path from the root of the cdag.

After the construction using the algorithm above, for all variables which are referenced in the basic block a cdag has been constructed. The root of this cdag is a node labeled with that variable. The cdag represents the computation of the value which should be in the variable at the end of the basic block.

6.2. Determining the evaluation order of the dag.

In the previous section we have constructed a dag for every basic block. For such a dag an evaluation order has to be determined. The *labeling algorithm* and *dynamic programming algorithm* [Aho et al. 1986], which can determine an evaluation order, are described. Then the labeling algorithm is adapted so that a better evaluation order can be determined when values are stored in registers at the beginning and end of a basic block, and when the basic block contains common subexpressions.

All these algorithms can not be directly applied to the MC68020, because it has two sorts of registers. How we can change these algorithms so that this is still possible is explained.

6.2.1. The evaluation order for trees.

We now have constructed a dag for every basic block, and have to determine the evaluation order from this dag. But determining the evaluation order from a dag is very difficult. Therefore we first try to determine the evaluation order for dags which happen to be trees. If a dag is a tree, determining the evaluation order is much easier because:
- Only one result is computed by every subtree.
- There are no shared nodes in the graph. (no common subexpressions)

At first we will also assume that all variables are stored in memory (on a stack) and none of them in a register at the beginning or end of a basic block.

We discuss two algorithms (both are described in [Aho et al. 1986]) which can determine an optimal evaluation order from a tree in an amount of time that is a linear function to the size of the tree, but with some assumptions on the instruction set of the target machine and without taking algebraic properties of operators into account. These algorithms are the *labeling algorithm* and the *dynamic programming algorithm*. Both have been used in a number of compilers. The requirements for the instruction set are described below.

6.2.2. The labeling algorithm.

The labeling algorithm [Aho et al. 1986] can determine an optimal evaluation order for a tree (without taking algebraic properties of operators into account) for certain register machines, for example for the following machine: (example from [Aho et al. 1986])
- n interchangeable registers.
- move, (dyadic) add and (dyadic) subtract instructions.
- absolute, register, register indirect and register indirect with displacement addressing modes.
- instruction costs are: 1 + the number of absolute and register indirect with displacement operands of the instruction.

The labeling algorithm can be used for machines in which all computation is done in registers and in which instructions consist of an operator applied to two registers or to a register and a memory location.

The algorithm consists of two phases.

During the first phase every node is *labeled* with a number. This number is the number of registers which are necessary to evaluate the subtree with as root this node without storing intermediate results in memory. (except for leaves which are not the leftmost child of its parent, see below) The nodes are labeled as follows.

If the node is a leaf, the node is labeled 1 if it is the leftmost child of its parent, otherwise it is labeled 0. This is because an operator can not compute its result (in a register) if the first argument is stored in memory and the second argument is stored in a register. But the operator can compute its result if the first argument is stored in a register and the second in memory. For commutative operators (like add) both computations are possible, but we don't take algebraic properties of operators into account, so the first computation is not allowed. Therefore the result of a leftmost child of its parent first has to be loaded into a register, so 1 register has to be used and the node is labeled 1. But for other leafs the result does not have to be loaded into a register first, so no registers have to be used and the node is labeled 0.

Otherwise, so if the node is an interior node, the node is labeled with the number of registers required to evaluate the subtree with as root this node in a register without storing intermediate results in memory. This can be done recursively by first calculating the label for every child, and than use these labels (numbers) to calculate the number of registers required to evaluate the subtree with as root this node. Let n_1, n_2, .., n_k

be the children of the node ordered so that label $(n_1) \geq$ label $(n_2) \geq \ldots \geq$ label (n_k). In the second phase these children will be evaluated in the order $n_1, n_2, .., n_k$ (see below). After every evaluation of a child one register is no longer available, because the result of the child is stored in it, so that label (n_i) + i - 1 is the number of registers used after evaluating $n_1, n_2, .., n_i$ (in that order). So the label of the node should become the maximum of label (n_i) + i - 1 for $1 < i \leq k$.

During the second phase we determine the evaluation order and generate code. To generate code for a tree, first code is generated recursively for all the *argument trees* of the root. With an argument tree of a node I mean a maximal subtree having as root a child of the node. The order in which the argument trees are evaluated is determined by the labels of the roots of the argument trees.
First code is generated for the argument tree with the highest label (which requires the highest number of registers for the evaluation), then for the argument tree with the second highest label (which requires the second highest number of registers for the evaluation), .. , and finally for the argument tree with the lowest label (which requires the fewest number of registers for the evaluation). If argument trees have the same label, the argument trees may be evaluated in any order. Finally code is generated to evaluate the whole tree using the results computed by the code for the argument trees. If the required number of registers for a subtree is higher than the number of available registers, the subtree is evaluated in memory, otherwise in a register.

For example to evaluate a-(b+c), the nodes are labeled:

```
a                       1
b                       1
c                       0
(b+c)                   max (1,1) = 1
a-(b+c)                 max (1,2) = 2
```

And the following code is generated:

```
        MOVE.L     a,D0
        MOVE.L     b,D1
        ADD.L      c,D1
        SUB.L      D1,D0
```

For example to evaluate a-((b+c)-(d+e)), the nodes are labeled:

```
a                       1
b                       1
c                       0
(b+c)                   max (1,0+1) = 1
d                       1
e                       0
(d+e)                   max (1,0+1) = 1
((b+c)-(d+e))           max (1,1+1) = 2
a-((b+c)-(d+e))         max (2,1+1) = 2
```

So only two registers are necessary, and when evaluating a-((b+c)-(d+e)) first ((b+c)-(d+e)) has to be evaluated, and then a, because a-((b+c)-(d+e)) is labeled 2 and a is labeled 1. The following code can be generated:

```
        MOVE.L     b,D1
        ADD.L      c,D1
        MOVE.L     d,D0
        ADD.L      e,D0
        SUB.L      D0,D1
        MOVE.L     a,D0
        SUB.L      D1,D0
```

6.2.3. The dynamic programming algorithm.

The requirements for the instruction set for the dynamic programming algorithm [Aho et al. 1986] to determine an optimal evaluation order from a tree (without taking algebraic properties of operators into account) are:

- The machine has n interchangeable registers.
- The machine has load, store and register to register copy instructions. Other instructions are of the form: register i = expression, where expression may contain operators, registers and memory locations. If the expression uses registers, register i has to be one of these registers.
- Every instruction may have its own instruction cost, but these costs should be fixed, i.e. may only depend on the instruction type and the addressing modes of its operands.

The dynamic programming algorithm results in a *contiguous evaluation*. A contiguous evaluation is an evaluation which always first evaluates all the subtrees that need to be computed in memory and then evaluates the remainder of the tree by evaluating the argument trees which need to be computed in a register one at a time, and finally evaluates the root of the tree. An example of a non contiguous evaluation is an evaluation which first partly evaluates subtree T1, then evaluates subtree T2 completely in a register and then evaluates the remainder of T1 in a register.

For a machine which meets the requirements described above we can prove an optimal contiguous evaluation always exists, i.e. a contiguous evaluation exists which is at least as fast as any not necessarily contiguous evaluation. So to determine an optimal evaluation order we only have to determine an optimal contiguous evaluation order.

The dynamic programming algorithm consists of three phases.

During the first phase for every node an array C of costs is calculated, where:
- C [0] is the minimal cost for evaluating the subtree with as root this node in memory without using registers.
- C [i] for i>0 and i≤number of available registers, is the minimal cost of evaluating the subtree with as root this node in a register, by using at most i registers.

The cost arrays are calculated recursively by first calculating the cost arrays for all children of a node, and then use these cost arrays to calculate the cost array of the node.

During the second phase the cost arrays are used to determine which subtrees should be evaluated in memory and which ones in a register.

During the third phase we traverse the tree, use the cost arrays to determine in what order the argument trees of a node should be evaluated and generate the code. The code for subtrees which should be evaluated into memory is generated first.

For example to evaluate `a-((b+c)-(d+e))` (same example as for the labeling algorithm) on an MC68020 with 3 available data registers, and instruction costs as the instruction execution times in appendix B, and access to variables using the address register indirect with displacement addressing mode are:
(evaluating an addition or subtraction node in memory without using registers is done by first storing a register in memory, then performing the computation using this register, and then reloading this register from memory, because the MC68020 can not compute the result of an addition or subtraction without using a register)

```
a                       C = { 0, 7, 7, 7 }
b                       C = { 0, 7, 7, 7 }
c                       C = { 0, 7, 7, 7 }
(b+c)                   C = { 30, 14, 14, 14 }
d                       C = { 0, 7, 7, 7 }
e                       C = { 0, 7, 7, 7 }
(d+e)                   C = { 30, 14, 14, 14 }
((b+c)-(d+e))           C = { 68, 51, 30, 30 }
a-((b+c)-(d+e))         C = { 75, 58, 37, 37 }
```

Again only two registers are necessary and the following code can be generated:

```
MOVE.L    b,D1
ADD.L     c,D1
MOVE.L    d,D0
ADD.L     e,D0
SUB.L     D0,D1
MOVE.L    a,D0
SUB.L     D1,D0
```

6.2.4. Using these algorithms to generate code for the MC68020.

Both algorithms described above can generate a good evaluation order in time linearly proportional to the size of the tree. The dynamic programming algorithm can be applied to a broader class of register machines, but is slower, specially when many registers are available.

Unfortunately the MC68020 machine model does not meet the requirements for these algorithms. The biggest problem is that the MC68020 does not have n interchangeable registers, but two classes of registers: address registers and data registers.

If we would ignore this problem, and deal with all registers in the same way, the code generated will not be good, because many computations can only be done in one sort of register, for example memory can only be addressed using address registers, and multiplications, divisions and shifts can only be done in data registers. If for example a computation has to be done using an address registers, and only a data register is available instead of an address register, we would have to move the value in an address register to the data register, do the computation using that address register and then move the value in the data register back to the address register.

To solve this problem we could adapt the labeling algorithm by calculating for every node:
- the number of data registers and the number of address registers required to evaluate the subtree with as root this node in a data register.
- the number of data registers and the number of address registers required to evaluate the subtree with as root this node in an address register.

By using this algorithm the code generator will be slower and more complicated, because now 4 numbers have to be calculated for every node instead of 1. By using this algorithm a good evaluation order can probably be determined, but the algorithm is not optimal, because if for example we can choose between using 2 data registers and 0 address registers or 1 address register and 1 data register we can't determine which choice is the best during the first phase.

The dynamic programming algorithm could also be adapted to solve the problems caused by the partitioning of registers into two classes, by calculating for every node the cost for returning a result in a data register for every possible combination of number of data registers and number of address registers and also the same costs for returning the result in an address register. So for every node we would have to calculate the costs to evaluate the subtree with as root this node:

- in memory without using registers.
- in a data register for every combination of at least one data register and any possible number of address registers.
- in an address register for every combination of at least one address register and any possible number of data registers.

By reserving the registers (A3-A7) and D7 as described in section 3.5, 3 address registers and 7 data registers remain available. Then we would have to calculate 1 + 7*4 + 3*8 = 53 costs for every node. We could leave out many of these costs for nodes which are the root of a small subtree, because then the costs for large numbers of registers are the same, because for small subtrees it doesn't matter if we may use for example 5, 6 or 7 registers. But even then would the number of costs per node be far too high, so that this algorithm would be too slow, very complicated and use too much memory. So this doesn't seem a good approach. But it would generate better code than the adapted labeling algorithm, because it doesn't have

problems when it has to choose between using for example 2 address registers and 0 data registers or 1 address register and 1 data register, because it can return both costs.

Another possibility is to combine both algorithms. Because the number of available address registers (3) is much lower that the number of available data registers (7), a good allocation of address registers is probably more important than a good allocation of data registers. It may therefore be sensible to calculate for every possible number of address registers the costs and the number of required data registers for every node. So for every node the following should be calculated:

- the cost of evaluating the subtree of which this node is the root.
- for every number of address registers the cost and number of required data registers to evaluate the subtree of which this node is the root in a data register.
- for every number of address registers greater than 0 the cost and number of required data registers to evaluate the subtree of which this node is the root in an address register.

So we would have to calculate 1+4+3 = 8 costs and 4+3 = 7 numbers of data registers for every node. Because 15 values have to be calculated for every node, this algorithm will probably be slow, complicated and use a lot of memory. But his algorithm will determine a better evaluation order than the adapted labeling algorithm. Although it is still not optimal, because we if we have to choose between using one data register more or slightly slower code, we can't determine which choice is the best during the first phase.

Consequently, we will use the adapted labeling algorithm because it can probably determine a good evaluation order and is faster, less complicated and uses less memory than the other two algorithms. But this algorithm has to be further adapted, because there are more problems. This is discussed below.

6.2.5. The evaluation order for dags with common subexpressions.

If a basic block contains common subexpressions, the dag constructed from this basic block will not be a set of trees, because the dag will contain shared nodes. Then the labeling algorithm can no longer be used. And determining the evaluation order becomes much more difficult. Generating optimal code from a dag for a one-register machine is NP-complete [Bruno et al. 1976]. Even with an unlimited number of registers the problem remains NP-complete [Aho et al. 1977].

According to [Aho et al. 1986] a reasonable solution can be obtained by partitioning the dag into a set of trees by finding for every shared node the maximal subtree with this shared node as root that includes no other shared nodes, except as leaves. This maximal subtree will be evaluated first, so that all common subexpressions will be evaluated first. So from a dag we obtain a set of trees, which can all be evaluated using an algorithm described in the previous section.

But if there are many small common subexpressions this solution results in a bad evaluation order. If for example many variables occur more than once in a dag, these variables are all common subexpressions. Using the solution of [Aho et al. 1986] we would first evaluate (load) all these variables in a register, so that we would probably have too few variables available to evaluate the remainder of the dag efficiently. To solve this we could evaluate some common subexpressions in memory, but it is not clear which common subexpressions should be evaluated in memory, and this will often not result in the best possible code.

Consequently, I don't think this is a good solution. I have tried to find other solutions for this problem in the literature, but I haven't found one.

Therefore I have tried to extend the labeling algorithm, so that it could determine a good evaluation order for dags with common subexpressions. This extension is described below, it doesn't take into account that there are data registers and address registers.

I will first explain what I mean with an *argument cdag*. An argument cdag of a node is a cdag (see section 6.1.9) with as root a child of the node. (compare argument tree of a node)

If we evaluate a shared node for the first time, we do not only have to compute the value of the expression represented by the cdag with as root this shared node, but we also have to remember this value, so that we

don't have to do the computation again if the shared node is used again. We will remember this value in a register. If the shared node is evaluated (encountered) again, we return this register. This register can be released after the last evaluation of this shared node.

So the number of used registers increases if a common subexpression is evaluated the first time and decreases if a common subexpression is evaluated the last time. So if we evaluate a cdag containing shared nodes (representing common subexpressions), the number of used registers may not only change because the result is returned in a register, but also because a shared node is evaluated.

Because evaluating shared nodes may change the number of used registers, the labeling algorithm will not always determine a good evaluation order. For example, if we want to compute the sum of expressions E1 and E2, and to evaluate E1 in a register requires 3 registers, and to evaluate E2 in a register requires 2 registers, then the labeling algorithm would generate code which first evaluates E1 and then E2. But if E1 contains common subexpressions so that the number of used registers increases by 3 if E1 is evaluated, and E2 doesn't contain common subexpressions, then it is better first to evaluate E2 and then E1. Because if we first evaluate E1 and then E2, then after evaluating E1 3 registers are no longer available and we need 2 registers for E2, so that we need 5 registers to compute the sum. But if we first evaluate E2 and then E1 we only need 4 registers to compute the sum.

To solve this we could extend the labeling algorithm by not only calculating for every node the number of registers required to evaluate the cdag with as root this node, but also by how many registers the number of used registers increases if the cdag is evaluated.

To calculate this increase in the number of used registers of a node we could use the labels of the children of the node and whether or not the node is a shared node. We could try to calculate this change in the number of used registers by adding the increase in the number of used registers of all the children of the node and by adding 1 if the node is a shared node.

Unfortunately this result will not be correct if the expression which is represented by the cdag with as root this node contains some common subexpressions of which all occurrences are in this expression. If for example the increase in the number of used registers is calculated for 'a+a' then 'a' is a common subexpression, so for 'a' the increase in the number of used registers is 1, and for 'a+a' we would calculate 1 + 1 = 2 as the increase in the number of used registers. But if 'a' is only used in this expression, we can release the register which contains the value of 'a' after having calculated 'a+a'. So the increase in the number of used registers for 'a+a' is not 2, but 0.

To solve this we could maintain for every node how many times every shared node occurs in the cdag with as root this node. This could be calculated during the first phase of the labeling algorithm. If we would also calculate how many times every shared node occurs in the whole dag, we could determine for every node the shared nodes of which all occurrences are in the cdag with as root this node, by comparing the number of occurrences in the whole dag with the number of occurrences stored in the node.

But storing this information for every node uses a lot of memory if there are a lot of common subexpressions, and makes the code generator slower and more complicated. The memory use is O (n*n) where n is the number of nodes, but usually much less memory is used. If we would not maintain this information, but simply use the sum of the increases in the number of used registers of all the children of the node plus 1 if the node is a shared node as the increase in the number of used registers of the node, the calculated increase will often be correct, but sometimes too high. For the implementation this approach was used, so the implementation does not calculate for every node how many times every shared node occurs in the cdag having as root this node. But the implementation does calculate how many times every shared node occurs in the whole dag, because we need this information to determine when a shared node is evaluated the last time. (see below)

After having calculated (an estimate of) the increases in the number of used registers and the required numbers of registers during the first phase of this adapted labeling algorithm, we can begin with phase two to determine an evaluation order by using these numbers. But after evaluating a shared node for the first time, the calculated increase of the number of used registers of this node will often no longer be correct, because the value of the expression which is represented by the cdag having as root this shared node is now in a register and does not have to be computed again if the shared node will be evaluated again. So the increase in the number of used registers when this shared node is evaluated is now 0. And also after

evaluating a shared node for the second last time the calculated increase of the number of used registers will often no longer be correct, because the value of the expression which is represented by the cdag having as root this shared node is now in a register and this register may now be released after the next use, because it is the last use. So then the increase is -1. (if the second last time is the same as the first time, the increase is also -1)
In both cases, the calculated increase of the number of used registers when the cdag having as root this node is evaluated is usually no longer correct. And usually the calculated number of required registers is no longer correct as well. Consequently, the increase in the number of used registers and the required number of registers for all nodes, for which the cdag with as root this node contains the shared node, may have changed. So in both cases these numbers have to be recalculated.

To recalculate these numbers when a common subexpression has been evaluated, we have to recalculate the numbers for the nodes for which the cdag with as root this node contains the shared node corresponding to the common subexpression. These nodes are the nodes for which there is a path to the shared node corresponding to the common subexpression. We could use the following algorithm:

```
PROCEDURE recalculate (NODE n)
      recalculate_node (n);
      IF increase in number of used registers after evaluating node n has changed
      OR required number of registers for node n has changed THEN
            FOR all parents p of node n DO
                  recalculate (p);
END_PROCEDURE recalculate.
```

Note that we have to be able to find the parents of a node, so we must store pointers to the parents of a node. This recalculating would make the code generator a lot slower if there are common subexpressions, but the evaluation order would be better. This has not (yet ?) been implemented in this implementation.

This extended labeling algorithm always completely evaluates the argument trees of a node one at a time. But there does not always exist such an optimal evaluation order. For example, if `D` is a dyadic function and `T` is a tryadic function and `T ( D (T (a,b,c),D (d,e)), D (T (c,a,b),D (f,g)), D (T (b,c,a),D (h,i)) )` has to be evaluated. Then `a`, `b` and `c` are common subexpressions which all occur three times. If `D` and `T` are operands of the target machine, we require 6 registers to evaluate this expression if we evaluate this expression by always completely evaluating the argument trees of a node one at a time. Because after evaluating one of the arguments of the outer `T` function four registers are in use, and we need an additional two registers to evaluate another argument. But if we first evaluate `T (a,b,c)`, `T (c,a,b)` and `T (b,c,a)` using five registers, we can evaluate the whole expression using only five registers. So this evaluation order will be better for certain machines.

This algorithm will also not always find the best evaluation order which always completely evaluates the argument trees of a node one at a time. This is not so strange, because I think this problem is also NP-complete.

If there are no common subexpressions, the amount of time which the algorithm uses to determine an evaluation order is O (n), where n is the size of the dag. But if there are many common subexpressions and recalculating the increases of the number of used registers and the required number of registers is performed, the amount of time which the algorithm uses is O (n*n).

6.2.6. The evaluation order for dags with variables in registers.

Up to now we have assumed variables are stored in memory at the beginning and end of a basic block. But if variables are stored in a register at the beginning or end of a basic block, the algorithm described above will often not determine a good evaluation order.

Because if a cdag is evaluated which evaluates (uses) a variable which is stored in a register at the beginning of the basic block for the last time, this register can be released, so the number of used registers decreases if such a cdag is evaluated. And if a cdag is evaluated which stores a result in a variable which has to be stored in a register at the end of the basic block, the number of used registers increases, because

the register which contains the variable is no longer available after the evaluation. The algorithm does not take these changes in the number of used registers into account.

But this can be solved easily, because these changes in the number of used registers resemble the changes in the number of used registers for common subexpressions.

If a variable is stored in a register at the beginning of a basic block, we can treat this variable in the same way as a common subexpression which has been evaluated once, because the value of a common subexpression has been stored in a register after the first evaluation and this register can also be released after the last evaluation. (use) This also means that if such a variable has been evaluated the second last time, we may have to adjust the increase in the number of used registers and the required number of registers for some nodes, just like when a common subexpression is evaluated the second last time, because these numbers of the node which represents the variable will have changed. This can be done in the same way as for common subexpressions as described in the previous section.

If a variable has to be stored in a register at the end of a basic block, we can treat this variable as a common subexpression which has not yet been evaluated, because if a common subexpression is evaluated the first time also a register has to be allocated to store the result. For a common subexpression this register can be released after the last evaluation (use), but for a variable which has to be stored in a register at the end of a basic block the register can not be released. We can solve this by pretending an extra use of such a variable exists in the basic block, so that the register will never be released.

But usually the variable may not be stored in any register at the end of a basic block, but has to be stored in a specific register. This causes problems if a cdag is evaluated which assigns a value to a variable which has to be stored in a specific register at the end of a basic block and that register has already been allocated. How this has been solved is described in section 6.4.7.

6.2.7. Calculation of the evaluation order of the arguments for a machine with one type of register.

After a dag has been constructed for a basic block, the dag consists of a set of cdags which may have shared nodes. These cdags will be evaluated completely one at a time. The order in which these cdags have to be evaluated has to be determined. And also the order in which the argument cdags of a node will be evaluated has to be determined. These orders are determined using the required number of registers and the increase in the number of used registers stored in the roots of the cdags.

Let n = the number of cdags to be evaluated. Assume the cdags are numbered from 1 to n, and that G (i) is the cdag with number i for all $i \in \{1, 2, .., n\}$, so that a cdag can be identified by its number using G.

For all cdags g let:

I (g) = the increase in the number of used registers when the cdag g will be evaluated, i.e. the number of registers in use after the evaluation of g - the number of registers in use before the evaluation of g.

U (g) = the (additional) number of registers necessary to evaluate the cdag g, i.e. the maximum number of registers in use during the evaluation of g - the number of registers in use before the evaluation of g. (often also called required number of registers)

After the graph has been constructed these I (g) and U (g) are calculated for every node in the graph during the first phase of the adapted labeling algorithm as has been partly described in sections 6.2.5 and 6.2.6.

The evaluation order of the cdags can be represented by a permutation P of {G (1), G (2), .., G (n)}, so that the graph to be evaluated first is P (G(1)), the graph to be evaluated second is P (G(2)), etc.

The maximum number of registers in use during the evaluation of P (G(i)), after evaluating P (G(1)), P (G(2)), .. ,P (G(i-1)) is:

$$R(P,i) = [\ \sum_{m=1}^{i-1} I(P(G(m)))\ ] + U(P(G(i)))$$

For all cdags g let D (g) = U (g) - I (g). Then

$$R(P,i) = [\ \sum_{m=1}^{i-1} I(P(G(m)))\ ] + U(P(G(i))) = [\ \sum_{m=1}^{i} I(P(G(m)))\ ] + D(P(G(i)))$$

The number of registers necessary to evaluate all the graphs in the order P (G(1)), P (G(2)), .., P (G(n)) is:
$R(P) = MAX\ \{\ R(P,i) \mid i \in \{1,2, ..,n\}\ \}$

To find the evaluation order which requires the minimum number of registers, we have to find a permutation P for which R (P) is minimal. Then the number of registers required is:
$R = MIN\ \{\ R(Q) \mid Q$ a permutation of $\{\ G(1), G(2), .., G(n)\ \}\ \}$

To find this evaluation order we could of course calculate R (P) for all permutations P, but this would cost an amount of time O (n!). For large n this amount of time is too high, therefore we have to find a faster method to determine this evaluation order. A method which can find the evaluation order by sorting is described below. This method can be implemented to execute in an amount of time of O (n log n), by using an O (n log n) sorting algorithm. For our implementation a simple sorting algorithm has been used, so that determining the evaluation order of a set of cdags costs an amount of time of O (n*n).

To use as few registers as possible, we should start evaluating the cdags, which evaluation results in the release of registers, because after evaluating such cdags, more registers will be available for the evaluation of other cdags.
The cdags, which evaluation does not result in a change in the number of used registers, should be evaluated when the highest number of registers is available, so after evaluating all the cdags which result in the release of registers.
Finally the cdags, which evaluation results in an increase in the number of used registers, should be evaluated, because all cdags which will be evaluated after this cdag will have fewer registers available, so these cdags should be evaluated as late as possible.

More formal, to use as few registers as necessary during evaluation:
- First evaluate all cdags g for which I (g) < 0.
- Then evaluate all cdags g for which I (g) = 0.
- Finally evaluate all cdags g for which I (g) > 0.

The cdags g for which I (g) < 0 have to be evaluated in an order so that the cdags for which U (g) is low are evaluated first. So the cdags for which I (g) < 0 have to be sorted from low to high using as ordering criterion U (g).
The cdags g for which I (g) = 0 may be evaluated in any order.
The cdags g for which I (g) > 0 have to be evaluated in an order so that the cdags for which D (g) is high are evaluated first. So the cdags for which I (g) > 0 have to be sorted from high to low using as ordering criterion D (g). That this ordering uses as few as possible registers is proved below.

So to use as few registers as possible:
- First evaluate all cdags g for which I (g) < 0, ordered from low to high by U (g).
- Then evaluate all cdags g for which I (g) = 0.
- Finally evaluate all cdags g for which I (g) > 0, ordered from high to low by D (g).

That the cdags have to be ordered in this way to use as few registers as possible is not difficult to understand, except that cdags for which I (g) > 0 have to be ordered from high to low by D (g). Therefore I will prove this below.

Theorem. Let E be a permutation of { G(1), G(2), .., G(n) } so that for all $k \in \{1,2, .. ,n-1\}$: D (E (G(k))) ≥ D (E (G(k+1))).
If for all cdags g: I (g) > 0 then $\mathtt{R} = R(E)$, i.e. E (G(1)), E (G(2)), .., E (G(n)) is an evaluation order which requires the minimal number of registers.

Proof. By definition: $R(E) = MAX \{ R(E,i) \mid i \in \{1, 2, .., n\} \}$, so $\exists m \in \mathbb{N} : R(E,m) = R(E)$.

Let $H = \{ E(G(l)) \mid l \in \{1, 2, .., m\} \}$ and let P be a permutation of {G(1), G(2), .., G(n)}, then there exists $k \in \mathbb{N}$ so that $k \geq m$, $P(G(k)) \in H$ and $\{ P(G(l)) \mid l \in \{1, 2, .., k\} \} \supseteq H$.

Then $R(P,k) =$ (by definition :)

$= [\sum_{l=1}^{k} I(P(G(l)))] + D(P(G(k)))$ (because $\{ G(l) \mid l \in \{1, 2, .., k\} \} \supseteq H$ and I (g)>0 :)

$\geq [\sum_{g \in H} I(g)] + D(P(G(k)))$ (because $P(G(k)) \in H$, $\exists l \leq m : E(G(l)) = P(G(k))$:)

$= [\sum_{g \in H} I(g)] + D(E(G(l)))$ (because $E(G(l)) \geq E(G(m))$:)

$\geq [\sum_{g \in H} I(g)] + D(E(G(m))) = R(E,m)$

Then $R(P) = MAX \{ R(P,i) \mid i \in \{1,2, .. ,n\} \} \geq R(P,k) \geq R(E,m) = R(E)$, and thus $R(P) \geq R(E)$.

Then for all permutations P: $R(P) \geq R(E)$, thus $\mathtt{R} = R(E)$.

For now we have assumed that I (g) and U (g) are fixed, i.e. evaluating a cdag does not change the I (g) and U (g) of other cdags. But evaluating a cdag may change the I (g) and U (g) of other cdags if:
- a common subexpression is evaluated the first or second last time.
- a value is stored in a variable which should be stored in a register at the end of the basic block.
- a variable which is stored in a register is evaluated the second last time.

In all these cases I (g) and U (g) of other cdags may decrease, but will never increase. The I (g) and U (g) can also be too high because during the first phase of the extended labeling algorithm the I (g) and U (g) are sometimes estimated too high. Consequently the I (g) and U (g) could be too high, but never too low.

Therefore we can improve the evaluation order by:

- Ordering the cdags for which I (g) = 0 so that the cdags for which U (g) is small are evaluated first. If for example `a` and `b` are cdags for which I (`a`) = I (`b`) = 0, U (`a`) = 2, U (`b`) = 3 and evaluating `a` results in a decrease of U (`b`) by one and evaluating `b` results in a decrease of U (`a`) by one, it is better first to evaluate `a` instead of `b`, because in that case we would need 2 registers instead of 3.

- Not only using D (g) as ordering criterion for cdags for which I (g) > 0, but also, if the D (g) of cdags are the same, use U (g) as (second) ordering criterion, and first evaluate the cdags for which U (g) is small. If for example `a` and `b` are cdags for which I (`a`) = 1, U (`a`) = 3, I (`b`) = 2, U (`b`) = 4 and evaluating `a` results in a decrease of U (`b`) by two and evaluating `b` results in a decrease of U (`a`) by two, it is better first to evaluate a instead of b, because in this case we would need 3 registers instead of 4.

For now we have only tried to determine an evaluation order for which the required number of registers is as low as possible. But if the required number of registers is higher than the number of available registers, some evaluation orders may be better than other evaluation orders which require the same number of registers. Because if not enough registers are available some values have to be stored in memory, and extra memory accesses are necessary, and for some evaluation orders the number of extra memory accesses could be smaller than for other evaluation orders.

Therefore we may improve the evaluation order by not only using U (g) as ordering criterion for cdags for which I (g) < 0, but also, if the U (g) of cdags are the same, use I (g) as (second) ordering criterion, and first evaluate the cdags for which I (g) is small ('more negative'). If for example U (a) = U (b) = 3, I (a) = -2 and I (b) = -1, where a and b are cdags, and one register is available before the evaluation of a and b, it is probably better first to evaluate a and then b, because then b can be evaluate without storing intermediate results in memory, because 3 registers are available after evaluating a. If however b is evaluated first, both a and b will have to store some intermediate results in memory.

After adding these extensions, to use as few registers as possible:

- First evaluate all cdags g for which I (g) < 0, ordered from low to high by U (g), and if the U (g) are the same, ordered from low to high by I (g).

- Then evaluate all cdags g for which I (g) = 0, ordered from low to high by U (g).

- Finally evaluate all cdags g for which I (g) > 0, ordered from high to low by D (g), and if the D (g) are the same, ordered from low to high by U (g).

6.2.8. Calculation of the evaluation order of the arguments for the MC68020.

Because the MC68020 has two types of registers we can't directly use the algorithm of the previous section. To still be able to use this algorithm for the MC68020 we could ignore the difference of data registers and address registers, by using the increase in the number of used data registers + the increase in the number of used address register as the increase in the number of registers for every node, and using the required number of data registers + the required number of address registers as the required number of registers for every node, and then apply the algorithm of the previous section to these numbers to determine the evaluation order of a number of cdags.

In this way we would treat address registers as being just as important as data registers. But by reserving the registers A3-A7 and D7 as described in section 3.5, 3 address registers and 7 data registers remain available. So address registers are more rare than data registers, because there are only 3 address registers available compared to 7 data registers. But both address and data registers are required often. Therefore, how address registers are allocated is much more important than how data registers are allocated. We could say an address register is 7/3 times as import as a data register. So we would like to treat an address register as being 7/3 times more important than a data register when determining the evaluation order, because the evaluation order determined in this way would probably be better.

We can adapt our algorithm to treat address and data registers in this way by using 3 * the increase in the number of used data registers + 7 * the increase in the number of used address register as the increase in the number of registers for every node, and using 3 * the required number of data registers + 7 * the required number of address registers as the required number of registers for every node, and use these numbers to determine the evaluation order.

The evaluation order determined in this way is not even optimal for the number of used registers if we don't take into account that evaluating a cdag may change the required number of registers and the increase in the number of used registers of other nodes. To obtain better results, we use a more accurate way of determining the evaluation order if we have to determine the evaluation order of only two cdags. This improves the evaluation order considerably because most nodes have only two argument cdags.

So assume the evaluation order of only two cdags has to be determined. Then there are only two possible evaluation orders: first evaluate the first argument cdag and then the second argument cdag, or first evaluate the second argument cdag and then the first argument cdag. For both evaluation orders we can determine the number of used address registers (UA) and data registers (UD), and the increase in the number of used address registers (IA) and data registers (ID). If l and r are the cdags to be evaluated, then:

- The number of used address registers for evaluating l first is MAX (UA (l), IA (l) + UA (r)) and the number of used data registers is MAX (UD (l), ID (l) + UD (r)).
- The number of used address registers for evaluating r first is MAX (UA (r), IA (r) + UA (l)) and the number of used data registers is MAX (UD (r), ID (r) + UD (l)).

How the increases in the number of used registers can be calculated has already been described in section 6.2.5, but these numbers will be the same for both evaluation orders. Using the numbers of used address registers and data registers we should determine in what order these two cdags have to be evaluated. But if we have to choose between using more data registers or using more address registers we don't know which choice is better. Again we can choose to treat address registers to be 7/3 times more important as data registers. So we will choose the evaluate order for which 3 * number of used data registers + 7 * number of used address registers is minimal.

The following examples illustrates that this may lead to a better evaluation order. Assume l and r are cdags and IA (l)=-1, ID (l)=1, UA (l)=0, UD (l)=1 and IA (r)=0, ID (r)=1, UA (r)=0, UD (r)=2. Then the improved way of determining the evaluation order would first evaluate r and then l, so that only 2 data registers are used and no address registers. But the other algorithm would first evaluate l and then r, so that 3 data registers and no address registers are used.

If the results of both cdags have to be used to compute a new result, the calculation of the required number of registers could be more complicated, but we can still use the same condition for choosing the evaluation order of the two cdags.

6.2.9. The evaluation order for dags with common subexpressions on the MC68020.

In the previous section we calculated the order of the arguments using for every argument cdag:
- the required number of data registers to evaluate the cdag. (UD)
- the required number of address registers to evaluate the cdag. (UA)
- the increase in the number of used data registers when the cdag is evaluated. (ID)
- the increase in the number of used address registers when the cdag is evaluated. (IA)

But in section 6.2.4 (before I introduced the problems with common subexpressions and variables in registers) we computed for every root of a subtree:
- the number of data registers required to evaluate the subtree in a data register.
- the number of address registers required to evaluate the subtree in a data register.
- the number of data registers required to evaluate the subtree in an address register.
- the number address registers required to evaluate the subtree in an address register.

So we computed both the register costs for evaluating the result in a data register and for evaluating the result in an address register. If we would also do this for our adapted labeling algorithm, we would have to compute UD, UA, ID and IA for evaluating the result in a data register and also for evaluating the result in an address register. So we would have to calculate 8 number of registers for every node. And we would no longer be able to determine the evaluation order of a number of cdags as described in section 6.2.8. Maybe both algorithms can be changed to handle this, but this would probably make them too complicated and the code generator would be too slow.

Therefore the following solution was chosen. Before the evaluation order is determined is decided which cdags should be evaluated in a data registers and which ones in an address register. How this is done is described in section 6.4.2. Then the first phase of the adapted labeling algorithm calculates for every node the register costs (UD, UA, ID and IA) for evaluating the cdag having as root this node in the required sort of register.

But because it usually is not necessary to evaluate all arguments in registers before an operation can be performed, for example for additions only one of the arguments has to be in a register, we can often not compute the register costs of such an operation from the register costs of the argument cdags. Therefore for some nodes we calculate the register costs to evaluate the cdag in memory, instead of the register costs to evaluate a cdag in a data register or an address register.

For the (original) labeling algorithm this was not a problem, because the labeling algorithm didn't take algorithm properties of operations into account. Therefore it could solve this problem by assuming the number of registers to evaluate a cdag, consisting of just a variable node, is one if it is a left leave and zero otherwise. (explained in section 6.2.2) But this often results in inefficient code for commutative operations like ADD. (commutativity is an algebraic property) For example, the labeling algorithm would evaluate A+(B+C) by first loading A in a register, then loading B in a register, adding C and then adding this result to the register containing A, instead of loading B in a register, then adding C and then adding A. This latter evaluation uses one register less, and consists of only 3 instructions instead of 4.

But even then we do not always have enough information to compute the register costs. Therefore for every node a flag is computed. This flag is true if a register may be released after accessing the result, otherwise false. If this flag is true, then:
- If the cdag is evaluated in a data register, this data register may be released after use.
- If the cdag is evaluated in an address register, this address register may be released after used.
- If the cdag is evaluated in memory, an address register may be released after use.

So for every cdag we compute:
- There the result of the cdag is computed, in an address register, in a data register or in memory.
- The required number of data registers to evaluate the cdag. (UD)
- The required number of address registers to evaluate a cdag. (UA)
- The increase in the number of used data registers when the cdag is evaluated. (ID)
- The increase in the number of used address registers when the cdag is evaluated. (IA)
- A flag indicating whether a register may be released after the result computed by the cdag has been used.

An example of such a calculation can be found in appendix A. (A.1.5)

6.3. The dag representation of the ABC instructions.

In the previous sections a dag was constructed for every basic block of ABC instructions and was explained how we could determine an efficient evaluation order of this dag. But how exactly this dag looks like has not yet been explained. This is done in this section.
So, for the ABC instructions is explained how they are represented in the dag and why this representation has been chosen. Also the nodes which are used for this representation are explained. These nodes can also be found in appendix C. Examples of these dags can be found in appendix A.

6.3.1. The sort of dag representation.

To be able to use the extended labeling algorithm described in 6.2 we first have to construct a dag from the ABC instructions. I already described how a dag could be constructed by representing every ABC instruction by a node in the graph in section 6.1.9. But such a representation has considerable disadvantages:

- Many different nodes for a similar operation. For example the following 13 ABC instructions test whether two values are equal: `eqB`, `eqB_a`, `eqB_b`, `eqC`, `eqC_a`, `eqC_b`, `eqI`, `eqI_a`, `eqI_b`, `eqR`, `eqR_a`, `eqR_b` and `eq_desc`. If we would like to optimize an operation, we would have to test for many different nodes. For example:
  - If we would like to optimize comparing to zero by using a `TST` instruction instead of a `CMP #0` instruction, for all 13 nodes the code generator would have to test for `#0` and have to be able to generate an extra code sequence which uses a `TST` instead of a `CMP #0`.
  - If we would like to optimize the not of a test for equality by using a `NE` (not equal) condition code after the `CMP` instruction instead of `EQ` (equal), the code generator would have to test for 14 different nodes when generating code for not.

- There are many complex nodes. For these nodes large codes sequences have to be generated, many code sequences are possible and complicated calculations are necessary to calculate the number of required registers of such a node.

Consequently it is better to use one node for similar operations, use nodes which represent simple computations, and can be implemented with only a few MC68020 instructions. So we would obtain a lower level representation, closer to the MC68020. Then performing optimizations is simpler, code sequences for nodes are shorter and calculations of the number of required registers are simpler. But the dag will be larger and translating the ABC instructions to a dag is more difficult.

For example, we would like to have only one node representing the operation which tests whether two values are equal, but the values to be compared are not always of the same type. Therefore two different nodes are necessary for this compare operation: CMP_EQ to compare integers and FCMP_EQ to compare floating point numbers.

These CMP_EQ and FCMP_EQ nodes have two arguments. Arguments could be the result of another computation, stored in memory, stored in a register or a constant. If an argument is the result of a computation the argument is represented by a cdag. If we would make nodes representing values stored in memory, values stored in a register and constants, these arguments can also be represented by cdags. Then all the arguments are cdags.

6.3.2. The dag representation for arguments.

An integer constant i is represented by a 'LOAD_I i' node, an address constant (label) l by a 'LEA l' node and a descriptor of symbol s and number of arguments a by a 'LOAD_DES_I s a' node.

The contents of an address or data register r is represented by a 'REGISTER r' node for registers which contain an intermediate result, or by a 'GREGISTER r' node for global registers, i.e. registers ASP, BSP, CSP, HP and FC, which may not be released after all uses in the basic block.
An integer on the B-stack in memory or an address on the A-stack in memory is represented by a 'LOAD d r' node. The value represented by this node is the contents before the execution of this basic block of the long word at the address computed by adding (integer constant) d to the address in stackpointer r at the beginning of the basic block.

Other integers and addresses stored in memory, i.e. the integers and addresses stored in the heap, in a descriptor element or in a symbol record, also have to be accessed. These integers and addresses are always accessed indirectly through a pointer and are represented by a 'LOAD_ID d g' node. The value represented by this node is the contents of the long word at the address computed by adding d to the value represented by cdag g.
We also have to be able to access bytes in memory, because the characters in a string are bytes. Bytes in memory are represented by 'LOAD_B_ID d g'. The value represented by this node is the zero extended contents of the byte at the address computed by adding d to the value represented by cdag g.
And we also have to be able to access words in memory, because the descriptors are words and the descriptor elements contains words. Words in memory are represented by 'LOAD_DES_ID d g'. The value represented by this node is the sign extended contents of the word at the address computed by adding d to the value represented by cdag g.

Using the nodes described above we can represent the arguments of all the instructions we want to represent in the dag, except for the instructions using floating point numbers.

6.3.3. The dag representation for operations.

Binary operations which don't use floating point numbers which can be represented in the dag by one node are: ADD, SUB, CMP_EQ, CMP_GT, CMP_LT, MUL, DIV, MOD, LSL, LSR, ASR, AND, OR and EOR. All these instructions can usually be implemented by one MC68020 instruction, but sometimes extra

`MOVE` instructions are necessary, and for the compare instructions sometimes extra instructions are necessary to convert a condition code to a boolean. The unairy CNOT node represents the not of a boolean.

Using the nodes we can represent the computation of the ABC instructions which we want to represent in the dag and don't use floating point values and don't access the heap.

6.3.4. The dag representation for floating point arguments.

In section 6.1.6 I already mentioned that representing instructions, which compute a floating point number, in a dag is more difficult because a floating point number consists of two long words. We can't always treat these two long words as one floating point number, because sometimes these two long words are manipulated separately, for example a floating point number is copied on the B-stack by two `push_b` instructions which each copy one long word.

To be able to represent the high and low long word of which a floating point number consists in the dag we use two nodes which point to a cdag which represents a floating point number. The 'FHIGH g' node represents the high long word of the floating point number represented by cdag g. And the 'FLOW g' node represents the low long word of the floating point number represented by cdag g.

If an instruction has a floating point number as argument, we could treat this floating point argument as two long words, so that for example a node representing a floating point addition would have four arguments. But if we would compute the two long words, of which a floating point argument consists, one at a time, the generated code would be inefficient. Because then, before we can perform the floating point operation, we would first have to make a floating point number out of the two long words. But usually the two long words are the high and low long word of the same floating point number. So we would not have to compute the high and low long word separately, but could compute the floating point argument at once and use this number for the floating point operation. But to do this the code generator would have to test for every floating point argument whether the two long words are represented by a FHIGH and a FLOW node having the same cdag as argument.

Consequently, a better representation can be obtained by representing a floating point argument by one cdag, so that for example a node representing a floating point addition would have two arguments. Then the FHIGH and FLOW are not necessary if the high and low long word of the floating point number are computed by the same cdag. To be able to still represent a floating point argument for which the high and low long word are computed by different cdags, we use a 'FJOIN g1 g2' node. A 'FJOIN g1 g2' node represents the floating point number consisting of the high long word represented by cdag g1 and the low long word represented by cdag g2.

Floating point arguments can now be represented just like integers and addresses. A floating point constant f is represented by a 'FLOAD_I f' node.

The contents of a floating point register fr is represented by a 'FREGISTER fr' node. And a floating point number stored in memory on the B-stack is represented by a 'FLOAD d r' node. The value represented by this node is the floating point number stored before the execution of the basic block in the two long words at the address computed by adding (integer constant) d to the address in stackpointer r at the beginning of the basic block.

Other floating point numbers stored in memory, i.e. floating point numbers stored in the heap, also have to be accessed. These floating point numbers are always accessed indirectly through a pointer and are represented by a 'FLOAD_ID d g' node. The floating point number represented by this node is the floating point number stored in the two long words at the address computed by adding d to the value represented by cdag g.

Using the nodes described above we can represent all the floating point arguments of all the instructions we want to represent in the dag.

6.3.5. The dag representation for floating point operations.

Binary operations using floating point numbers which can be represented in the dag by one node are: FADD, FSUB, FCMP_EQ, FCMP_GT, FCMP_LT, FMUL, FDIV and FREM. Unairy operations using floating point numbers which can be represented in the dag by one node are: FACOS, FASIN, FATAN, FCOS, FEXP, FITOR, FLN, FLOG10, FRTOI, FSIN, FSQRT and FTAN. All these instructions can usually be implemented by one MC68881 instruction, but sometimes extra `FMOVE` instructions are necessary.

Using these nodes we can represent the computation of the ABC instructions which we want to represent in the dag and which use floating point values and don't access the heap.

6.3.6. The dag representation for storing values on the stack and in registers.

To represent a store of an integer or address in memory we use a 'STORE d r g1 g2' node. This node represents that the long word at the address computed by adding (integer constant) d to the address in stackpointer r at the beginning of the basic block should contain the integer or address represented by cdag g1 after execution of the basic block. What cdag g2 is used for is explained in section 6.4.7.

To represent a store of an integer or address in a register we use a 'STORE_R r g' node. This node represents that register r should contain the integer or address represented by cdag g after execution of the basic block.

To represent a store of a floating point number in memory we use a 'FSTORE d r g1 g2 g3' node. This node represents that the two long words at the address computed by adding (integer constant) d to the address in stackpointer r at the beginning of the basic block should contain the floating point number represented by cdag g1 after execution of the basic block. What cdags g2 and g3 are used for is explained in section 6.4.7.

There is no representation for storing a floating point number in a register, because in the current implementation floating point numbers are always stored in memory at the beginning and end of a basic block.

6.3.7. The dag representation for push_args and repl_args.

As I already explained in section 6.1.6, representing the `push_args` and `repl_args` ABC instruction in the dag causes problems, because the result of these instructions is not one value on the A-stack or B-stack, but multiple values on the A-stack. To represent these instructions we could construct a node in the dag for every value on the A-stack. A possibility is to represent each argument using a 'LOAD_ID' node.

For example a push_args which pushes the first four arguments of the node represented by cdag g1 on the A-stack will then be represented by:

	'LOAD_ID 0 g2'	for the first argument,
	'LOAD_ID 4 g2'	for the second argument,
	'LOAD_ID 8 g2'	for the third argument,
	'LOAD_ID 12 g2'	for the fourth argument and
g2:	'LOAD_ID 8 g1'	for the variable size part of the node containing the arguments.

But if we would generate code for a push_args instruction represented in this way, we would not be able to use an MC68020 `MOVEM` instruction to move all four arguments to registers with one instruction, or use the address register indirect with postincrement addressing mode to address the arguments, (except the last one) but have to use the address register indirect with displacement addressing mode to address the arguments (except the last one), which results in less efficient code.

For example if code is generated for a `push_args` with four arguments, and the address of the node is in address register `A0`, the generated code could be:

```
MOVEA.L    8(A0),A1                ; load pointer to variable size part of
                                     the node containing the arguments.
MOVE.L     (A1),D0                 ; load first argument in D0
MOVE.L     4(A1),D1                ; load second argument in D1
MOVE.L     8(A1),D2                ; load third argument in D2
MOVE.L     12(A1),D3               ; load fourth argument in D3
```

But if we would use the `MOVEM` instruction, we could generate the following code:

```
MOVEA.L    8(A0),A1                ; load pointer to variable size part of
                                     the node containing the arguments.
MOVEM.L    (A1),D0-D3              ; load four arguments in registers D0, D1,
                                     D2 and D3
```

According to the cache case execution times in appendix B this code is 3 percent faster, and 4 words long instead of 9 words, so this code is better.

If we would use the address register indirect with postincrement addressing mode, we could generate the following code:

```
MOVEA.L    8(A0),A1                ; load pointer to variable size part of
                                     the node containing the arguments.
MOVE.L     (A1)+,D0                ; load first argument in D0
MOVE.L     (A1)+,D1                ; load second argument in D1
MOVE.L     (A1)+,D2                ; load third argument in D2
MOVE.L     (A1),D3                 ; load fourth argument in D3
```

According to the cache case execution times in appendix B this code is 10 percent faster than the first example, and 6 words long instead of 9 words, so this code is better than the first example.

To be able to use a `MOVEM` instruction or the address register indirect with postincrement addressing mode, a different representation was chosen. Every value pushed on the A-stack by a `push_args` or `repl_args` instruction is represented by a 'MOVEMI g1' node. For every value argument cdag g1 of the 'MOVEMI g1' node is the (same) cdag with as root a 'MOVEM d1 g g1 .. gn' node. This 'MOVEM d1 g g1 .. gn' node represents n values stored at n consecutive long words at the address computed by adding (integer constant) d1 and the address represented by cdag g. Cdags g1, .., gn are the cdags with as root a MOVEMI node, i.e. g1 is the cdag with as root the MOVEMI node representing the first argument, g2 is the cdag with as root the MOVEMI node representing the second argument, etc. Although this introduces cycles in the dag, we do not change out terminology.

For example a push_args which pushes the first four arguments of the node represented by cdag g1 on the A-stack will be represented by:

g4:	'MOVEMI g3'	for the first argument,
g5:	'MOVEMI g3'	for the second argument,
g6:	'MOVEMI g3'	for the third argument,
g7:	'MOVEMI g3'	for the fourth argument,
g3:	'MOVEM 8 g2 g4 g5 g6 g7'	representing all 4 arguments and
g2:	'LOAD_ID 8 g1'	for the variable size part of the node containing the arguments

Using this representation the code generator can use a `MOVEM` instruction or the address register indirect with postincrement addressing mode. How this is done is explained in section 6.4.9.

6.3.8. The dag representation for create and del_args.

To create a new node in the heap, the following actions have to be taken:
- subtract the size of an empty node (3 long words) from register FC and if the value in this register is less than zero after this subtraction call the garbage collector.
- compute the address of the new node.
- fill the new node with the descriptor and the address of the reduction code, and add the size of the new node to register HP.

Subtracting the size of an empty node from register FC and eventually calling the garbage collector is not represented in the dag. How this is done is explained in section 6.9.

Two possible representations for a `create` instruction are:
- Use several cdags, each representing a part of creating a new node. One cdag to represent the address of the new node, one to store the descriptor in the heap, and one to store the address of the reduction code in the heap.
- Use one cdag, which represents the address of the new node, and storing the descriptor and the address of the reduction code in the heap, and adding the size of the node to HP.

If we would use the representation with several cdags, we would use the heap pointer HP just like the A and B-stack pointers. We would have to do a simple heap simulation just like the stack simulations for the A and B-stack and we would need a STORE node to store a value in the heap just like for storing a value on the A-stack or B-stack. Then creating a node would be represented by the following three cdags:

```
1:          STORE -4 ASP g1          ; store the address of the new node on the A-stack
      g1:   GREGISTER HP             ; the address of the new node is in HP
2:          STORE 0 HP g2            ; store the address of the reduction code in the heap
      g2:   LEA cycle_in_spine       ; the address of the reduction code of an empty node
3:          STORE 4 HP g3            ; store the descriptor in the heap
      g3:   LOAD_DES_I empty 0       ; the descriptor of an empty node
```

But this representation has the following disadvantages:
- We can't use the address register indirect with postincrement addressing mode or a `MOVEM` instruction to store the values in the heap, which is more efficient than using an address register indirect with displacement addressing mode, just like for the `push_args` and `repl_args` instruction (see section 6.3.7)
- We can't change the location of nodes in the heap created by create instructions. If for example two nodes are created using two create instructions, and it is more efficient to do the second create first, a node is created at address 12+HP instead of at the address in HP, which means we first have to add 12 to HP before we can use the postincrement addressing mode using HP to store the values of the node in the heap. If we would be able to reverse the location of these two nodes, this addition is not necessary.

The representation using one large cdag does not have these disadvantages. Therefore a create is represented in this way by a cdag having as root a 'CREATE g1 g2 g3' node. A 'CREATE g1 g2 g3' node represents the address of a new node in the heap which consists of the three long words represented by cdags g1, g2 and g3. If code is generated for such a CREATE node, the new node is stored at the address currently in HP and the size of the new node is added to HP.

So creating a node is represented by the following cdag:

```
(       STORE -4 ASP g1          ; store the address of the new node on the A-stack)
g1:     CREATE g2 g3 g4          ; the address of a new empty node
g2:     LEA cycle_in_spine       ; the address of the reduction code of an empty node
g3:     LOAD_DES_I empty 0       ; the descriptor of an empty node

g4:                              ; don't care value, third long word of an empty node is not used
```

The ABC instruction `del_args` also creates a new node, but this is not an empty node, but a copy of another node with some arguments deleted. Because nodes may share arguments in this implementation, the arguments do not have to be copied. We can therefore represent a `del_args` with a 'CREATE g1 g2 g3' node, where cdags g1 and g2 represent the address of the reduction code and descriptor just as for the `create` instruction, and cdag g3 represents the address of the variable size part containing the arguments of the node from which this node is a copy with some arguments deleted.

6.3.9. The dag representation for fill instructions and set_entry.

Two possible representations for a fill instruction are:
- Use several cdags, each representing a store of one long word in the heap, for example by representing each store by a 'STORE_ID d r g' node.
- Use one cdag, which represents the address of the node, and filling the whole node.

Using several cdags, each representing a store of one long word, has the same disadvantages as using several cdags for a `create` instruction (see 6.3.8). Therefore we use one cdag to represent a fill instruction. A fill is represented by a 'FILL g g1 .. gn' node, where cdags g1, .., gn represent the long words which have to be stored in this order in n consecutive long words at the address represented by cdag g. The result of a 'FILL g g1 .. gn' node is the address represented by cdag g.

Using this FILL node we can fill integer, character, boolean and string nodes and nodes without arguments. To fill a node having arguments we also have to create a new variable size part in the heap containing the arguments, which can not be done with a FILL node. We already have a node to represent the creation of a new empty node, a CREATE node. But this CREATE node can only represent the creation of a node consisting of three long words. But if we allow a CREATE node to have any number of arguments, we can use this node to represent both the creation of a variable size part of a node containing arguments and the creation of a new node. Then a fill of a node having arguments can be represented by a FILL node, of which the fourth argument cdag (the third long word to be filled, the argument pointer) has as root a CREATE node which represents the arguments.

The same representation could also be used to fill floating point nodes, then the CREATE would have two argument cdags with a FHIGH and FLOW as root. But during code generation for a CREATE node we would then have to test for this case, because it is more efficient to move a floating point number to the heap at once, in stead of moving the two long words of which the floating point number consists separately. Therefore we use a special node for creating the variable size part of a floating point node in the heap, a 'CREATE_R g' node, so that we don't have to test for this case. A 'CREATE_R g' node represents the address of a new variable size part of a floating point node in the heap which consists of the floating point number represented by cdag g. If code is generated for a CREATE_R node, the floating point number is stored at the address currently in HP and the size of the variable size part of a floating point node is added to HP.

6.3.10. The dag representation for add_args.

The instruction `add_args` copies a node and adds some arguments to this copy. Because we don't know the size of the variable size part of the node at compile time, we can't represent this variable size part with a CREATE node, so we need a new node, the ALLOCATE node, to be able to represent `add_args`.

An 'ALLOCATE gc ga g1 .. gn' node represents the address of a new variable size part consisting of a number of long words from another node, represented by cdag gc and ga, where cdag gc represents the number of long words and ga represents the address of the variable size part of the other node, and the long words represented by cdags g1, .., gn. If code is generated for an ALLOCATE node, the long words are stored at the address currently in HP and the size of the long words is added to HP.

6.4. Generating intermediate code from the dag.

We now have represented the ABC instructions in the dag and have determined the evaluation order using the first phase of the adapted labeling algorithm. So we can now generate code from this dag using the second phase of the adapted labeling algorithm.
In this section is explained how *intermediate code* is generated from this graph. This intermediate code is very close to MC68020 machine code. But in this intermediate code an unlimited number of address and data registers may be used. How the code generator determines whether to use a data register or an address register of the MC68020, if a register is used, is described. Also is explained how the following optimizations are performed: optimizations of the creation of nodes by `create` instructions, optimizing the use of booleans by using condition codes, optimizing the use of small constants and how generating unnecessary store instructions is prevented.

6.4.1. Calculating reference counts.

To generate code from the dag the code generator has to know how many times a node is referenced, i.e. how many pointers point to it. This information is necessary because:

- For REGISTER nodes we have to know when we may release the register, so that it can be used for other purposes. This can be done by maintaining a counter, which counts the number of uses. After every use the counter is decremented and if it becomes zero the register may be released. To initialize these counters, we have to know how often this REGISTER node is referenced.

- Shared nodes have to be recognized. These are the nodes which are referenced more than once. Because if such a shared node is evaluated the first time, the result of the evaluation of the cdag has to be remembered, so that when this value is referenced again, it does not have to be recomputed. Such a value is remembered in a register. Therefore the shared node is overwritten by a REGISTER node the first time it is evaluated. And to initialize the counter of this new REGISTER node, we have to know how often the shared node is referenced.

Two ways in which we can calculate these counters are:
- Maintain the counters during dag construction.
- Construct the dag and then calculate the reference counters.

To maintain the counters during dag construction, we first have to initialize all counters with zero.
Then if a pointer to a node is added, do:

```
PROCEDURE adjust counters after adding a pointer (pointer)
      Increment the counter of the node to which the pointer points.
      IF the counter = 1 THEN
            FOR all arguments, of the node to which the pointer points, which
               are pointers DO
                  adjust counters after adding a pointer (argument).
END.
```

But also sometimes pointers are deleted. If for example a value is popped from a stack using a `pop_a` or `pop_b` instruction, the pointer to the cdag representing a value which is popped, has to be deleted. If a pointer is deleted, some reference counts have to be adjusted. This can be done by:

```
PROCEDURE adjust counters after deleting a pointer (pointer)
      Decrement the counter of the node to which the pointer points.
      IF the counter = 0 THEN
            FOR all arguments, of the node to which the pointer points, which
               are pointers DO
                  adjust counters after deleting a pointer (argument).
            /* Delete the node */
END.
```

To calculate the counters after dag construction, we first have to initialize the counters with zero. And then call the `PROCEDURE adjust counters after adding a pointer` for every cdag on the A-stack and B-stack. Every such cdag represents the computation of one value on the A-stack or B-stack.

Our implementation calculates the counters after dag construction, because:
- the counters are not needed during dag construction.
- it is faster than maintaining the counters, because now the counters never have to be decremented because a pointer is deleted.
- the `PROCEDURE adjust counters after deleting a pointer` does not have to be implemented.

A disadvantage of this method is that we can not delete nodes which are no longer referenced during dag construction.

6.4.2. Choosing between address registers and data registers.

When generating code for an operation we sometimes can choose between using an address register or a data register. For example if we have to add two values which are stored in memory, we can compute the result in an address register or in a data register, and both computations of the sum are equally efficient. If this sum has to be multiplied later, we should compute the sum in a data register, because the MC68020 multiply instruction can multiply a value in a data register, but not in an address register. But if this sum is the address of a value, and the value is referenced later, we should compute the value in an address register, because we can address the value using an address register indirect addressing mode, but we can't address the value if the address is stored in a data register. (If not enough registers are available this could be false)

To be able to make a good choice when we can choose between an address register and a data register, a counter is added to every node. This counter is initialized with zero. If a node is added to the dag, and better code can be generated for the operation represented by this node if the argument is computed in an address register instead of a data register, the counter in the root of this argument cdag is incremented. This is for example done for the second argument of a LOAD_ID node. But if better code can be generated if the argument is computed in a data register instead of an address register, the counter in the root of this argument cdag is decremented. This is for example done for both arguments of MUL and DIV nodes.

If we can choose between using an address register or a data register when generating code for a node, we use the counter in the node. If the counter in the node is greater than zero we use an address register, and if the counter is less than zero we use a data register. If the counter is zero, we prefer a data register, because after reserving the registers as described in section 3.5 7 data registers remain available, but only 3 address registers.

This method will usually give good results, but is not optimal, because:

- A counter is only incremented or decremented if the value needs to be in a specific sort of register to generate efficient code for a parent node, but not if the value needs to be in a specific sort of register to generate efficient node for some other node, for example a grand-parent node. For example for the cdag LOAD_ID 0 (ADD (ADD (LOAD 0 BSP) (LOAD 4 BSP)) (LOAD 8 BSP)) the cdag ADD (LOAD 0 BSP) (LOAD 4 BSP) will be computed in a data register, but should be computed in an address register, because after adding 8(BSP) the value will still be in a data register, but should be in an address register for LOAD_ID.

- If not enough registers are available, choosing a different sort of register is sometimes better. If for example not enough address registers are available, but enough data registers are available, it is sometimes better to do a computation which can be done more efficiently using address registers if enough address registers are available, by using data registers.

- A value is always stored in only one sort of register. If a value is needed more than once in both an address register and a data register, it is sometimes better to store the value both in an address register and a data register.

6.4.3. Register allocation during code generation from the graph.

During code generation from the dag we assume an unlimited number of address registers and an unlimited number of data registers are available. Why this is done will be explained in section 6.6 on local register allocation. Later the intermediate code will be changed so that only 8 address and 8 data registers are used, this is also explained in section 6.6.

But because we know the MC68020 doesn't have an unlimited number of registers, we try to generate code which uses as few registers as possible during code generation from the dag.

If during code generation a new register of a specific sort (address or data register) is required, the free register of that sort having the lowest number is allocated. If a register contains a value which is no longer used, the register is released, so that it can be allocated again. The registers A3-A7 and D7 are never allocated or released, because they contain stackpointers, etc. , as described in section 3.5.

6.4.4. The intermediate code.

The intermediate code is very similar to MC68020 code. The instructions of the intermediate code can be found in appendix D. I already said that in the intermediate code an unlimited number of address and data registers may be used. The other differences are:

- The address register indirect addressing mode does not exist. Instead the address register indirect with displacement addressing mode with displacement zero is used.
- The `MOVEA`, `ADDA`, `SUBA`, `ADDI`, `SUBI`, `CMPA` and `CMPI` instructions are simply called `MOVE`, `ADD`, `SUB` or `CMP` instructions.
- There are no `MOVEQ`, `ADDQ` and `SUBQ` instructions.
- There are no `BRA` and `BSR` instructions. Instead `JMP` and `JSR` are used.
- There is no `CLR` instruction.
- The `SEQ`, `SGE`, `SGT`, `SLE`, `SLT`, `SNE`, `FSEQ`, `FSGE`, `FSGT`, `FSLE`, `FSLT` and `FSNE` instructions of the intermediate code compute a long word, but the Scc instructions of the MC68020 compute a byte.
- The `BMOVE` instruction moves a block of long words. There is no such MC68020 instruction.
- `MOVE.L`, `ADD.L`, `SUB.L`, etc. are simply called `MOVE`, `ADD`, `SUB`, etc. `MOVE.B` and `MOVE.W` are called `MOVEB` and `MOVEW`.

The address register indirect addressing mode, the `MOVEQ`, `ADDQ`, `SUBQ`, `BRA`, `BSR` and `CLR` instructions are used by the final MC68020 code, because some instructions of the intermediate code are later optimized, for example a `MOVE #1,D0` instruction is optimized to a `MOVEQ #1,D0`. These optimizations are described in section 6.10.

6.4.5. Generating intermediate code from the dag.

After the reference counts have been computed, we can start generating intermediate code. For every element on the A-stack and B-stack for which a cdag has been constructed code has to be generated. The order in which these cdags are evaluated is determined as described in sections 6.2.7 and 6.2.8.

Then code is generated for these cdags by the following algorithm:
- First evaluate all the argument cdags of the root by recursively executing this algorithm. The argument cdags are evaluated in the order as described in sections 6.2.7 and 6.2.8.
- Generate code for the operation in the node using the values computed by the argument cdags.
- Return how the result can be addressed, and a counter which counts how many times the result will be used.

Which code is generated depends on:
- How the values computed by the argument cdags have to be addressed and how many times they will be used. This information is returned when code is generated for an argument cdag.
- Whether the node is shared. If it is, the result of the computation specified by this node is always stored in a register.
- Whether it is more efficient to return the computed value in a data register or an address register.

The value computed by a cdag can be returned:
- in a data register. (`Dn`)
- in an address register. (`An`)
- in a floating point register. (`FPn`)
- as an integer constant. (`#n`)
- as a descriptor. (`#n`)
- as a floating point constant. (`#n`)
- in a memory location of which the address is the sum of a signed 16 bit integer constant and an address in an address register. (`d(An)`)

6.4.6. Generating code for arithmetic dyadic operation nodes.

The result of arithmetic dyadic operations is always computed in a register, because computing the result in memory and later using this result in memory is always more expensive than computing the result in a register and later using the result in this register.

For example, assume we have to add one to a value in memory, and this value in memory will no longer be used after this addition, then we can compute the result in memory using:

```
ADD         #1,d(An)
```

And we can compute the result in a register using:

```
MOVE        d(An),Dr
ADD         #1,Dr
```

If we later use the value in memory, for example for another addition, we will use:

```
ADD         d(An),Dn
```

And if we later use the value in a register:

```
ADD         Dr,Dn
```

According to the cache case execution times in appendix B computing and using the value in a register is 45 percent faster for this example than computing and using the value in memory.

Only if the result has to be stored in the same memory location as one of the arguments has to be loaded from, computing the value in memory is often faster. For example if we have to generate code for the cdag STORE 4 BSP (ADD (LOAD 4 BSP) (LOAD 1)) we could generate:

```
ADD         #1,4(BSP)
```

instead of:

```
MOVE        4(BSP),Dr
ADD         #1,Dr
MOVE        Dr,4(BSP)
```

According to the cache case execution times in appendix B the first instruction sequence is executed 55 percent faster. But computing a result, which has to be stored in the same memory location as one of the arguments has to be loaded from, happens seldom, because values are often stored in registers. So the code

will usually be only slightly less efficient if we never compute the result in memory. Therefore we will always compute the result in a register.

To compute the result in a register, first the arguments are evaluated. For an arithmetic operation like ADD we then test if one (or both) of the arguments has been computed in a register of the right sort, i.e. the sort (address or data) of register the result of this ADD has to be computed in, and if this is the last use of this register, we generate an instruction which adds the other argument to this register. Otherwise we allocate a new register of the right sort, generate an instruction which moves one of the arguments to this register, and an instruction which adds the other argument to this register.

For other arithmetic dyadic operations code is generated in a similar fashion, but is sometimes more difficult. For example for non commutative operations, like SUB, and instructions for which the operands may not be stored in an address register, like MUL.

6.4.7. Generating code for STORE and FSTORE nodes.

To generate code for a 'STORE d1 r1 g1 g2' node, first code is generated for the argument cdag g1 which computes the value to be stored. Then the value can usually be stored in memory on the A-stack or B-stack using one MOVE instruction to d1(r1). But sometimes this causes problems, because storing the value in memory on the A or B-stack overwrites the value of d1(r1), and if this value will be used later, we will no longer be able to compute it.

Assume this value will be used later. We can solve the problem by allocating a register r2 and moving the value in memory location d1(r1) to this register using a MOVE instruction, before we move the value computed by the code for argument cdag g1 into this memory location, and for the later uses use the value in this register.

If the value of d1(r1) will be used later, a LOAD or FLOAD node exists, with a reference count greater than zero, which loads this value. To be able to test for this when generating code for a STORE node, argument cdag g2 of a STORE node points to this LOAD or FLOAD node if it exists. If such a LOAD node exists, the LOAD node is overwritten by a 'REGISTER r2' node, so that the later uses will use the previous value of d1(r1) stored in register r2, instead of the value which is now stored in this memory location. And also if such a FLOAD node exists, the FLOAD node is overwritten by a FREGISTER node, for the same reason.

Code for FSTORE is generated in a similar fashion, but because a floating point number consists of two long words, the FSTORE node contains two pointers which could point to a LOAD or FLOAD node, and it may be necessary to move two values to registers.

6.4.8. Generating code for STORE_R nodes.

To generate code for a 'STORE_R r1 g1' node, first code is generated for the argument cdag g1 which computes the value to be stored. If this value has been computed in register r1 no code has to be generated for the STORE_R node. Otherwise this value can usually be moved to register r1 using one MOVE instruction. But just as for the STORE node, this sometimes causes problems, because storing the value in a register overwrites the current value of the register, and if this value will be used later, we will no longer be able to compute it.

Assume the value in register r1 will be used later. To solve the problem, we can use a similar solution as for the STORE node, and allocate a register r2, and move the contents of register r1 to register r2 using a MOVE instruction, before we move the value computed by the argument of the STORE_R node into register r1.

If the value in register r1 will be used later, a 'REGISTER r1' node exists, with a reference count greater than zero. To be able to test for this when generating code for a STORE_R node, we can't use a similar solution as for the STORE node and store a pointer to this REGISTER node in the STORE_R node, because during code generation from the dag sometimes new REGISTER nodes are created. Therefore for

every register a pointer is maintained during construction of the dag and code generation from the dag, which points to the REGISTER node for this register if it exists. If a 'REGISTER r1' node exists with a reference count greater than zero, the register number of this node is overwritten by register r2, so that the later uses will use the previous value of register r1, instead of the value which is now stored in register r1.

If the value to be stored was computed in a register r3 when generating code for argument cdag g1, and register r1 contains a value which will be used later, then we would generate two `MOVE` instructions which move a value from one register to another. But if the value which is stored in register r1 may also be stored in an other register, we can generate more efficient code by using an `EXG` instruction. Because if we store the value, which register r1 contains before the store, in register r3, then what is currently stored in r1 has to be moved to r3, and what is currently stored in r3 has to be moved to r1. So we can generate code for the STORE_R node by exchanging the values in register r3 and register r1 by one `EXG` instruction, which is faster than two `MOVE` instructions, and changing the register number in the 'REGISTER r1' node into register r3.

6.4.9. Generating code for MOVEM and MOVEMI nodes.

Three possible ways to generate code for MOVEM and MOVEMI nodes are:
- address every long word using an address register indirect with displacement addressing mode.
- address every long word using an address register indirect with postincrement addressing mode.
- use a `MOVEM` instruction to move multiple long words to registers with one instruction.

Using the address register indirect with postincrement addressing mode is usually better than using the address register indirect with displacement addressing mode, because a postincrement addressing mode is executed faster and uses less memory than a displacement addressing mode. But a disadvantage of using the postincrement addressing mode is that the long words have to be moved starting with the first long word, then the second, then the third, etc. ,while for the displacement addressing mode we can move the long words in any order. Another disadvantage of the postincrement addressing mode is that it changes the address register, so that if the value has to be used later, we first have to copy the address in another register before we can move the long words.

If n long words have to be moved from memory, the cache case execution time (see appendix B) of the `MOVEM` instructions is 10+4*n clock cycles, and for using the postincrement addressing mode 6*n clock cycles. So when moving more than 5 long words `MOVEM` is faster, and when moving 5 long words, both instruction sequences are equally fast. But the `MOVEM` instruction always occupies 2 words and the instructions using the postincrement addressing mode occupy n words. So to move 5 long words it is better to use the `MOVEM` instruction.

But the `MOVEM` instruction can only move long words to registers in the order D0, D1, .., D7, A0, A1, .., A7, so that for example we can not move the first long word to an address register and the second long word to a data register using one `MOVEM` instruction.

Code is generated for a 'MOVEMI g1' node by first generating code for the argument cdag g1, which has a MOVEM node as root. After this code has been generated, the MOVEMI node will have been overwritten by a 'REGISTER r' node, and register r contains the result of this MOVEMI node and is returned.

For a 'MOVEM d1 g g1 .. gn' node code is generated by first generating code to compute the argument cdag g in an address register. Let Am be this address register. Then n registers are allocated and the long words in consecutive memory locations at the address `d1(Am)` are moved to these registers. If only one long word has to be moved, a `MOVE` from `d1(Am)` is used to move this long word into a register. If two long words have to be moved, and `d1` is not equal to zero or this is not the last use of the address in address register `Am`, two `MOVE` from `d(Am)` instructions are used. If at least 5 long words have to be moved, a `MOVEM` from `d1(Am)` instruction is used if possible. Otherwise, if this is the last use of the address in address register `Am`, `d1` is added to `Am`, else the address `d1(Am)` is loaded into a free address register, let `Am` now be this address register. And then n-1 `MOVE` from `(Am)+` instructions and a `MOVE` from `(Am)` instruction is used to move the long words from memory into the registers. Then all the MOVEMI nodes, which are the roots of the cdags g1, .., gn, are overwritten by 'REGISTER r' nodes, where r is the register in which the value represented by the MOVEMI node was stored.

6.4.10. Generating code for FILL nodes.

Just like for MOVEM nodes, three possible ways to generate code for FILL nodes are:
- address every long word using an address register indirect with displacement addressing mode.
- address every long word using an address register indirect with postincrement addressing mode.
- use a `MOVEM` instruction to move multiple long words from registers with one instruction.

Using the address register indirect with postincrement addressing mode is usually better than using the address register indirect with displacement addressing mode. The reasons for this have already been discussed in the previous section on the MOVEM node.

If n long words have to be moved to memory, the cache case execution time (see appendix B) of the `MOVEM` instructions is 6+3*n clock cycles, and for using the postincrement addressing mode 4*n clock cycles. So when moving more than 6 long words `MOVEM` is faster, and when moving 6 long words, both instruction sequences are equally fast. But the `MOVEM` instruction always occupies 2 words and the instructions using the postincrement addressing mode occupy n words. So to move 6 long words it is better to use the `MOVEM` instruction.

But the `MOVEM` instruction can only move long words from registers in the order D0, D1, .., D7, A0, A1, .., A7. Because the values to be moved are computed in an arbitrary register, in memory or are constants, it is unlikely that 6 or more consecutive values are computed in registers in such an order that they can be moved to memory using one `MOVEM` instruction, therefore we will never use the MOVEM instruction.

Code is generated for a 'FILL g g1 .. gn' node by first generating code for all argument cdags. Then the results computed by cdags g1, .., gn are moved to consecutive long words at the address computed by cdag g. If only one long word has to be moved, the address computed by cdag g is loaded into an address register `Am` and a `MOVE` to `(Am)` is used to move this long word into memory. If two long words have to be moved and cdag g has computed the address in an address register `Am` and this is not the last use of the address in `Am`, a `MOVE` to `(Am)` and a `MOVE` to `4(Am)` instruction are used. Otherwise, if cdag g has not computed the address in an address register or this is not the last use of this address, an address register is allocated and the address is loaded into this address register. Let `Am` be the address register which contains the address computed by cdag g, then n-1 `MOVE` to `(Am)+` instructions and a `MOVE` to `(Am)` instruction are used to move the long words into memory. For the argument cdags g1, g2, .., gn which are empty cdags no long word has to be stored in memory.

The result computed by a FILL node is the address of the node which is to be filled, which is the address computed by arguments cdag g. If the node is filled using the postincrement addressing mode, the address is changed during filling the node. So that if the address computed by the FILL has to be used later, this address has to be copied before filling the node. Usually this address is copied into a register. But often this address is only used later by a STORE node to store the address in memory on the A-stack. In such a case the address is first moved to a register, and then moved from this register to memory, but this address could have been moved to memory immediately, which is more efficient. In this case the reference count of such a FILL node is 1 and the parent is a STORE node, so we can implement this by testing if argument cdag g1 of a 'STORE d1 r1 g1 g2' node has a FILL node as root with a reference count 1 when generating code for a STORE node.

6.4.11. Generating code for CREATE nodes.

The code for a CREATE node has to fill consecutive long words in memory in the heap (a node), at the address in register HP, add the size of this node to register HP and return the address of this node. How a node should be filled has already been discussed for the FILL node in the previous section. For the CREATE node this is done in a similar way using the address register indirect with postincrement addressing mode. If this is performed using register HP, we don't have to use an extra instruction to add the size of the node to HP after the fill. To compute the address of the node, we could use a `LEA -d(HP),An` instruction after the fill, where d is the size of the node. But this can be done more efficient with

a MOVE from HP to a register before the fill, because such a `MOVE` is faster than a `LEA` instruction, and the address of the node could be stored using any addressing mode and not only in an address register.

So code is generated for a CREATE node by first generating code for all the arguments cdags, then allocating a register and generating a `MOVE` from HP to this register, and finally instructions are generated which move the long words computed by the argument cdags to memory using `MOVE` to `(HP)+` instructions. For the argument cdags of the CREATE node which are empty cdags no long word has to be stored in memory, but only the size of a long word has to be added to HP.

Just like for the FILL node, storing the address of the node in a register is not very efficient if the address is only used later by a STORE node to store the address on the A-stack, which happens often. Because in such a case the address is first moved from HP to a register, and then moved from this register to memory, but could have been moved from HP to memory immediately. Just like for the FILL node we can implement this by testing if argument cdag g1 of a 'STORE d1 r1 g1 g2' node has a CREATE node as root with a reference count 1 when generating code for a STORE node.

6.4.12. Optimizing the creation of nodes.

In section 5.1.1 I already mentioned that initialization of a node by a create instruction is not necessary if this node will always be filled before a garbage collection could occur. Many of these initializations are removed in the following way. Every time a FILL node is created the code generator tests if the first argument cdag of this FILL node, which computes the address of the node (or part of the node) which is to be filled, has a CREATE node as root and if the number of long words which are created by this CREATE node in the heap equals the number of long words which are filled by the FILL node. If this is so, all the argument cdags of the CREATE node are replaced by empty cdags. Because no value is stored in memory for empty argument cdags of a CREATE node, no initialization takes place for a CREATE node with only empty argument cdags.

Another optimization which was described in section 5.1.1, creating and filling a node at the same time, can be implemented in the following way . When code is generated for a FILL node, the code generator tests if the first argument cdag of the FILL node has a CREATE node as root, and if the reference count of this CREATE node is one, and if the number of long words which are created by this CREATE node in the heap equals the number of long words which are filled by the FILL node. It this is so, then the create and fill are done at the same time by not generating code for the 'FILL g g1 .. gn' node and the CREATE node, but generating code for a 'CREATE g1 .. gn' node.

6.4.13. Using the condition codes of the MC68020.

In section 5.2.4 I already mentioned that a conditional jump which has to be taken depending on a boolean which is the result of a comparison can be optimized. This can be done by using the condition codes of the MC68020 for the conditional jump, instead of first computing a boolean value and using this boolean value as condition for the jump. I also mentioned that computing the not of a comparison can often be optimized into a comparison (without not) by using the reverse condition code, for example by using 'NE' instead of 'EQ'.

This has been implemented by using a procedure which generates code for a cdag which computes the boolean represented by this cdag in a condition code and a procedure which generates code for a cdag which computes the not of the boolean represented by this cdag in a condition code. These two procedures are used to generate code for the conditional jump instructions, i.e. the ABC instructions `jmp_true` and `jmp_false`, and the CNOT node.

These two procedures generate a `CMP` or `FCMP` instruction for CMP_EQ, CMP_LT, CMP_GT, FCMP_EQ, FCMP_LT and FCMP_GT nodes. For a CNOT node they call the other procedure to generate code of the argument cdag of the CNOT node, and then return the reverse condition. For the remaining nodes code is generated which computes the result in a register, in memory or as constant as usual, and then this result is compared with zero using a `TST` or `CMP` instruction to obtain a (NE or EQ) condition code.

6.4.14. Preventing unnecessary stores.

Because a cdag, with usually as root a STORE node, is constructed for every value on the A-stack or B-stack which is referenced in the basic block, often cdags are constructed which just load a value in memory on the A-stack or B-stack and then store the value at the same address, i.e. 'STORE d1 r1 (LOAD d1 r1) g2' cdags. If for example an integer on the B-stack in memory is added to the top of the B-stack (using a `push_b` and an `addI` instruction), then for this integer such a cdag would be constructed. But no code has to be generated to store this integer in memory, because the integer has not been changed and was already stored in this memory location.

So for such a cdag an unnecessary `MOVE d1(r1),d1(r1)` would be generated. To prevent this we could test if argument cdag g1 of a 'STORE d1 r1 g1 g2' node consists of a 'LOAD d1 r1' node when generating code for a STORE node, and not generate any code in such a case.

But this would not prevent all unnecessary stores, because the result of a cdag having a FILL node as root is the same as the result of the first argument cdag of this FILL node. Consequently for cdag 'STORE d1 r1 (FILL (LOAD d1 r1) g1 g2 g3) g4' the address in d1(r1) would be loaded into a register, and then this address in this register would be stored in d1(r1), but d1(r1) already contains this address, so this store is not necessary.

To prevent these unnecessary stores and the ones described above, the following algorithm is used to generate code for a 'STORE d1 r1 g1 g2' node:

```
g := g1
WHILE the root of cdag g is a 'FILL g3 g4 .. gn' node DO
      g := g3
IF the root of cdag g is a 'LOAD d2 r2' node AND d2=d1 AND r2=r1 THEN
      an unnecessary store has been found, don't generate code for this STORE
      node, but generate code for argument cdag g1 of this STORE node
ELSE
      generate code for this STORE node as usual
```

6.4.15. Optimizing the use of small constants.

In section 2.3.7 I mentioned that instead of doing an operation (like for example `CMP` or `ADD`) with a long word constant between -128 and 127 as operand using an instruction with an immediate long addressing mode, it is usually faster and shorter first to move the value in a data register using `MOVEQ` and then do the operation. For example instead of:

```
CMP       #100,D0
```

it is faster and shorter to use:

```
MOVE      #100,D1
CMP       D1,D0
```

This could be implemented by testing for an immediate operand between -128 and 127 when generating intermediate code for nodes like `CMP` and `ADD` and then allocating a new register `Dn`, generate a `MOVE #i,Dn` instruction (which will later be optimized to a `MOVEQ` instruction), generate an instruction which performs the operation using register Dn instead of the immediate value and then release register `Dn`.

But this instruction sequence has as disadvantage that it uses an extra register. So that if not enough registers are available, using such an instruction sequence could result in extra load and store instructions during local register allocation (explained later in section 6.6). Then the code would be far less efficient then when this 'optimization' had not been done. Therefore this optimization should only be done if it doesn't result in extra load and store instructions.

This has been implemented by only performing this optimization if a parameter data register (one of the registers D0-D7, see section 6.6) is available to store the immediate value. If such a register is available, usually no extra load and stores are necessary during local register allocation (but not always). And if such a register is not available, then the number of registers in use is higher than the number of available registers, so in that case it is very likely that extra loads and stores will be generated during local register allocation. In the current implementation this has only been implemented for the `CMP` instruction, because comparing to small constants happens very often during pattern matching.

This optimization is also used when allocating memory in the heap (see section 6.9).

6.5. Global register allocation.

Before the dag for a basic block is constructed, the code generator determines which values are stored in registers at the beginning of the basic block. And after the dag has been constructed, which values are stored in registers at the end of the basic block. This is called *global register allocation*. Conditions for global register allocation are described. And a straightforward global register allocation algorithm which was used for this implementation is described. Also is explained how parameters and results of functions are passed.

6.5.1. Directives describing the parameters and results for the code generator.

In section 5.2.2 we concluded that it is usually better to pass parameters and results of functions in registers instead of in memory. But to be able to pass parameters and results in registers, the code generator has to know what kind of parameters a function has and what kind of result it computes. This information can not always be derived from the ABC code. Therefore the Clean compiler inserts directives in the ABC code, which describe the parameters and results when entering and leaving a function.

Because the parameters and results are passed on the top of the A-stack and B-stack, the parameters and results can be described by describing the stack layout of the top elements of the A-stack and B-stack which are parameters or results at certain locations in the program.

If a function is called using an ABC `jsr` instruction, the stack layout before and after the `jsr` instruction is described using a `.d(emanded)` directive before the `jsr` instruction and a `.o(ffered)` directive after the `jsr` instruction:

```
.d              #A_entries #B_entries B_types
jsr             function
.o              #A_entries #B_entries B_types
```

where:

`#A_entries` = The number of A-stack entries.

`#B_entries` = The number of B-stack entries. Floating point numbers are considered to be two entries, because they consist of two stack elements.

`B_types` = For every entry on the B-stack, starting with the entry on top of the stack, a character indicating the type of the entry:

- i = integer
- b = boolean
- c = character
- r = real (floating point number) (only one r for a floating point number, not two)

If a function is called using an ABC `jsr_eval` instruction, always only one parameter is passed on the A-stack and only one result is returned on the A-stack. So no directives are necessary to described the stack layout for a `jsr_eval` instruction, because the stack layouts are always the same.

If a function is jumped to using an ABC `jmp` instruction, the stack layout before the `jmp` instruction is described using a `.d` directive before the `jmp` instruction:

```
      .d              #A_entries #B_entries B_types
      jmp             function
```

If a function is jumped to using an ABC `jmp_eval` instruction, always only one parameter is passed on the A-stack. So no directives are necessary for a `jmp_eval` instruction, because the stack layout is always the same.

If a function is left using an ABC `rtn` instruction, the stack layout before the `rtn` instruction is described using a `.d` directive before the `rtn` instruction:

```
      .d              #A_entries #B_entries B_types
      rtn
```

And finally for every label, which is the entry point of a function, the stack layout is described using a `.o` instruction before the label:

```
      .o              #A_entries #B_entries B_types
label: no_op
```

6.5.2. Function calling convention.

Because passing parameters in registers is usually more efficient (see section 5.2.2), we will try to pass parameters in registers. But the MC68020 has two sorts of registers, so we also have to determine which parameters should be passed in which sort of register.

Parameters and results which are passed on the A-stack are addresses of nodes. Because only address registers can address memory, and therefore address the values in a node, these parameters and results can best be passed in address registers.

Parameters and results which are passed on the B-stack are integers, booleans, characters and floating point numbers. Integers, booleans and characters can best be passed in data registers, because the operations on these types can often be performed faster if the operands are in data registers than if they are in address registers.

Floating point numbers can probably best be passed in the floating point registers of the MC68881 coprocessor, because floating point operations can only be performed in these registers, but this has not yet been implemented. Currently they are passed in memory, which is better than passing them in address or data registers. This is because a floating point number has to be stored in two MC68020 registers, and the MC68881 can not perform a floating point operation using a floating point number stored in two MC68020 registers, but can perform a floating point operation using one operand which is stored in two consecutive long words in memory.

When registers have been reserved for stackpointers etc. as described in section 3.5, registers D0-D6 and A0-A2 can be used to pass parameters and results of functions. So only 3 address registers, 7 data registers and 8 floating point registers are available for passing parameters and results. Sometimes we will not have enough registers available to pass all parameters and results in a register. If not enough registers of a specific sort are available, only (as many as possible) parameters or results which are closest to the top of the stack are allocated a register of the required sort, the other parameters or results are passed in memory.

If some parameters or results can not be passed in data registers, and some address registers are still available, it may be more efficient to pass as many of these parameters or results in these address registers. And if some parameters or results can not be passed in address registers, and some data registers are still available, it may be more efficient to pass as many of these parameters or results in these data registers. This has not been implemented.

If n parameters or results are passed in registers of a specific sort, registers 0, 1, .., n-1 are used, and the parameter or result closest to the top of the stack is passed in register n-1, the parameter or result second closest to the top of the stack is passed in register n-2, etc. Why this order has been chosen will be explained in section 6.5.5.

If a function is called, jumped to or left, the stackpointers should point to the top of the stack. But if some elements on top of a stack are passed in registers, the stackpointer can not point to the top of the stack, because the top element of the stack is not stored in memory. Therefore the stackpointers point to the stack element closest to the top of the stack which is not passed in a register.

If we don't take precautions, the code for a function could change the values in registers, so that a register may no longer contain the same value after a `jsr` or `jsr_eval` as it contained before the `jsr` or `jsr_eval`. Consequently we should either:
- not assume a register contains the same value after a function call, or
- save all registers which may be changed by a function at the start of the code for a function and restore these registers before the function is left, or
- a combination of both methods, i.e. assume some registers may be changed by a function and some registers may not be changed by a function.

The convention I have used in this implementation is that a function may change the values of all registers, because:

- If we would assume a function may not change the value of registers, then at the beginning of a function we wouldn't know which registers contain values. This causes problems for the garbage collector, because the garbage collector has to be able to find all pointers to nodes. Because these pointers may be stored in address registers, the garbage collector has to know which address registers contain pointers to nodes, but the garbage collector can not derive this information. This could be solved by storing zero in an address register if it doesn't contain a value, but this would make the code slower.

- A Clean function has three entry points, but often only one exit point. If we would assume a function may not change the value of registers, then at all three entry points the same registers would have to be saved. This is inefficient for entry points for which only a few registers have to be saved.

6.5.3. Conditions for global register allocation.

As I already said, the global register allocator determines which values should be located in registers at the beginning and end of a basic block. The *beginning of a basic block* is the location immediately before the first instruction of the basic block. The *end of a basic block* is the location immediately after the last instruction of the basic block if this last instruction is an instruction without side effects, otherwise, so if last instruction of the basic block is an instruction with side effects, the end of the basic block is the location immediately before this instruction.

Which values are located in registers at the beginning or end of a basic block is called the *(global) register allocation*. But in order to obtain correct code, the global register allocation has to meet certain conditions.

Because if control may flow directly from basic block A to basic block B, the global register allocation at the end of basic block A should be the same as the global register allocation at the beginning of basic block B. So the A-stack and B-stack elements which are located in a register at the end of basic block A should be the same as the A-stack and B-stack elements which are located in a register at the beginning of basic block B, and the elements should also be located in the same registers. (This condition is a bit too severe. If basic block B pops a stack element from the stack and does not use this element, the register allocation for this stack element does not have to meet the condition. But because the implemented code generator can not detect this, we will ignore this.)

If the jsr_eval instruction is optimized as described in section 5.2.6, doing global register allocation is different than when this optimization is not performed. Because the implemented code generator performs this optimization, I assume this optimization is performed when describing global register allocation.

Consequently, for the ABC code the global register allocation has to meet the following conditions:

1. If control may flow directly from basic block A to the next basic block B, i.e. if the last instruction of basic block A is not an rtn, jsr, jmp , jsr_eval or jmp_eval instruction, the register allocation at the end of basic block A has to be the same as the register allocation at the beginning of basic block B.

2. If the last instruction of a basic block A is a jmp label, jsr label, jmp_false label or jmp_true label instruction, the register allocation at the beginning of the basic block which is labeled with label label should be the same as the register allocation at the end of basic block A.

3. If the last instruction of a basic block is a rtn instruction, the register allocation at the end of this basic block has to be the same as the register allocation:
   - At every location just after a MC68020 JSR instruction generated for a jsr_eval instruction which may call the function from which this rtn returns.
   - At the beginning of every basic block to which the rtn instruction may return to. These beginnings of basic blocks to which a rtn instruction may return to usually are all the basic blocks of which the previous basic block has as last instruction a jsr instruction.

4. If the last instruction of a basic block is a jsr_eval instruction, the register allocation at the end of this basic block has to be the same as the register allocation at the beginning of the next basic block, and the register allocation at the location just before the MC68020 JSR instruction generated for this jsr_eval instruction should be the same as the register allocation at the beginning of every basic block which may be jumped to by this jsr_eval instruction.

5. If the last instruction of a basic block is a jmp_eval instruction, the register allocation at the end of this basic block has to be the same as the register allocation at the beginning of every basic block which may be jumped to by this jmp_eval instruction.

6.5.4. Consequences of the function calling convention for global register allocation.

By defining the parameters and result by the function calling convention described in section 6.5.2, we have already defined (fixed) which values should be stored in memory and which values in registers at several locations in the program. Because at every location where a .d or .o directive is placed, the global register allocation should be as defined by the function calling convention.

So, we have already defined the global register allocation for:

- The beginning of every basic block which is labeled with a label which is an entry point of a function. Because there is a .o directive before such a label.
- The end of every basic block of which the last instruction is a jsr, rtn or jmp to a label, which is the label of a function entry point, instruction. Because there is a .d directive before these instructions.
- The beginning of every basic block of which the previous basic block has as last instruction a jsr instruction. Because there is a .o directive after this instruction.

And because of the conditions for global register allocation described in the previous section, we have also defined the global register allocation for:

- The end of every basic block of which the last instruction is an instruction without side effects of which the next basic block is labeled with an entry point of a function. Because of condition 1.
- Every location just before a MC68020 JSR instruction generated for a jsr_eval instruction. Because jsr_eval always calls an entry point of a function, and the register allocation of all entry points has been defined and because of condition 4.

- Every location just after a MC68020 `JSR` instruction generated for a `jsr_eval` instruction. Because the register allocation just before a `rtn` instruction has been defined for all `rtn` instructions, and because of the first case of condition 3.
- The end of every basic block having a `jmp_eval` as last instruction. Because `jmp_eval` always jumps to an entry point of a function, and the register allocation of all entry points has been defined and because of condition 5.

6.5.5. Straightforward global register allocation.

So by defining the function calling convention, we have defined the global register allocation at many locations in the program. But for the other beginnings and ends of basic blocks we still have to determine a global register allocation. For example at *local labels* and at `jmp_false` and `jmp_true` instructions. With a local label I mean a label which does not label a location which is the entry point of a function.

Because implementing an advanced global register assignment algorithm costs a lot of time, a straightforward global register assignment algorithm has been implemented, which gives reasonable results.

This straightforward global register allocation algorithm allocates registers at the locations described in the previous section as defined by the function calling convention and the conditions of section 6.5.3.

And to be able to determine register allocations for the other locations, so that the register allocations meet the conditions of section 6.5.3, the algorithm saves (defines) the global register allocation for a local label the first time the label is used. If a local label is encountered again, it uses this saved register allocation.

The global register allocation for the beginning of a basic block (except for the locations described in 6.5.4) is determined with the following rules:

- If the basic block is labeled by a local label for which a global register allocation has already been defined, registers are allocated according to this definition. In such a case this is not the first use of this label, and usually the label has already been used by a `jmp`, `jmp_true` or `jmp_false` instruction. Then this register allocation has to be chosen, because of condition 2 of section 6.5.3.

- Otherwise, if the previous basic block exists and the last instruction of the previous basic block is:
  - A `jmp`, `jmp_eval` or `rtn` instruction, then no registers are allocated, i.e. all stack elements are stored in memory. This will usually not happen.
  - An instruction without side effects, then registers are allocated in the same way as they were allocated at the end of the previous basic block. This is necessary because of condition 1 of section 6.5.3.
  - An instruction with side effects (but not `jmp`, `jmp_eval` or `rtn`), then registers are allocated as they are allocated after execution of the instruction with side effects.

- Otherwise, no registers are allocated. This will usually not happen.

If a register allocation at the beginning of a basic block is determined using any of these rules, the global register allocation of all the labels which label this basic block are defined to be this global register allocation.

Then the global register allocation for the end of a basic block (except for the locations described in 6.5.4) is determined with the following rules:

If the last instruction of the basic block is:
- A `jmp`, `jmp_true` or `jmp_false` instruction:
  - If the global register allocation of the label to which these instructions (may) jump has already been defined, then registers are allocated according to this definition. This is necessary because of condition 2 of section 6.5.3.

- Otherwise, we allocate registers for the top A and B-stack elements for which a register was allocated at the beginning of the basic block, or which are used or computed by this basic block. The registers are allocated in the same way as when passing parameters for functions and results of functions.
  The reasons for this allocation are:
  - If a register is allocated for a stack element at the beginning of the basic block, and no register would be allocated for the stack element at the end of the basic block, then an extra instruction would be necessary to store the value of the stack element in memory.
  - If a value is computed by this basic block, it can be computed faster in a register than in memory.
  - If a value is used by this basic block, it has to be loaded from memory. We load it into a register, because it is likely that the next basic block uses this value again. Then the next basic block doesn't have to load it from memory again, but can use the register.

  And then the global register allocation of the label to which the `jmp`, `jmp_true` or `jmp_false` instruction (may) jump is defined to be this global register allocation.

- An instruction with side effects (but not `jmp`, `jmp_true` or `jmp_false`), registers are allocated in such a way that the instruction with side effects can be executed.

- An instruction without side effects, then:
  - If the next basic block exists and the next basic block is labeled by a label for which a global register allocation has been defined, registers are allocated according to this definition. This is necessary because of conditions 1 and 2 of section 6.5.3.
  - Otherwise, we allocate registers just as for a `jmp`, `jmp_true` or `jmp_false` to a label of which the global register allocation has not yet been defined. So we allocate registers for the top A and B-stack elements for which a register was allocated at the beginning of the basic block, or which are used or computed by this basic block. The registers are allocated in the same way as when passing parameters for functions and results of functions.

If we would allocate the element on top of the stack to register 0, the second element of the stack to register 1, etc., we would often have to move values in registers to other registers. If for example 6 data registers have been allocated at the beginning of the basic block and code has to be generated for the following basic block:

```
pushI           10
eqI_b           1000 1
jmp_true        l1
```

and no global register allocation has been defined for label `l1`, then we would allocate 7 stack elements in data registers at the end of this basic block, and the code generator could generate the following intermediate code:

```
MOVE        D5,D6
MOVE        D4,D5
MOVE        D3,D4
MOVE        D2,D3
MOVE        D1,D2
MOVE        D0,D1
MOVE        #10,D0
CMP         #1000,D1
BEQ         l1
```

But if we would allocate the element on top of the stack to the register with the highest register number which contains a value of a stack element, and the second element of the stack to the register with the second highest register number which contains a value of a stack element, etc., as described in section 6.5.2, then the code generator would generate the following (much better) code:

```
MOVE        #10,D6
CMP         #1000,D5
BEQ         l1
```

Therefore this way of allocating registers has been used.

6.6. Local register allocation.

Because the intermediate code generated from the dag may use an unlimited number of data and address registers, and the MC68020 only has 8 data and address registers, a *local register allocator* changes the intermediate code of a basic block so that no more than 8 address registers and data registers are used. It does this by changing the register numbers of the registers used by the intermediate instructions and by inserting instructions to load and store values in registers from/into memory. This is explained below.

6.6.1. Local register allocation strategy.

Because the MC68020 has only 8 address registers and 8 data registers, we will sometimes not be able to retain all intermediate results in a register and therefore we will sometimes have to store intermediate results in memory by storing registers in memory and loading registers from memory.

The normal labeling algorithm (see section 6.2.2) determines when to store registers in memory and load registers from memory during code generation from the tree. (phase 2) Suppose code has to be generated for a tree having an 'OP g1 g2' node as root, where OP is the dyadic operation represented by this node and g1 and g2 are the argument subtrees which represent the computations of the operands. If the required number of registers for the evaluation for both g1 and g2 are higher than the number of available registers, then g2 is evaluated first, then the register in which the result of g2 has been computed is stored in memory, then g1 is evaluated and finally the result of OP is computed using the result of g2 in memory and the result of g1 in a register. This method gives very good results, often optimal.

Unfortunately using this method for cdags which contain shared nodes does not give good results, because:

- The calculated required number of registers to evaluate a cdag is often too high if the cdag contains common subexpressions. Consequently, a register may be stored in memory and loaded from memory, although enough registers are available to evaluate the cdag without storing intermediate results in memory.

- This method can not determine whether the result of a shared node after the first evaluation, i.e. the result of a common subexpression, should be stored in a register or in memory. We can't store all results of shared nodes in registers, because there is a limited number of registers, and storing all results of shared nodes in memory would result in inefficient code. We could try to store the results of shared node in registers, and if later not enough registers are available, move one of the registers containing the result of a shared node to memory. But then we would not be able to make a good choice between these registers, because to determine which register should be stored in memory, we would have to know when these registers will be used again, which we do not yet know at that time.

Therefore register allocation was not done during code generation from the dag, but was postponed till after the code for a basic block had been generated by assuming an unlimited number of address registers and data registers are available. Because then we would be able to determine when a register will be used again in this basic block, which was one of the reasons why we could not determine a good register allocation during code generation from the dag.

To describe the strategy we have used, we will call the (possibly too many) registers which have been allocated during the code generation from the dag *virtual registers*, and call the registers of the target processor *real registers*. If a virtual register is used, a real register is allocated and this real register is used as this virtual register. The strategy we have used allocates the real register containing the value which will not be used for the longest time and stores this register in memory if a real register has to be allocated and all real registers are in use, i.e. contain values which will be used later. And if a virtual register is used of which the value has been stored in memory, a real register is allocated and the value is loaded from memory

and this real register is used as this virtual register. This strategy was shown to be optimal in a page swapping context in [Belady 1966].

Because the MC68020 has two sorts (data and address) of registers, we have to adapt this strategy. The value of a virtual register always has to be stored in a real register of the same sort, because otherwise we may obtain instructions with invalid operands, for example a `MULS.L A1,A0` instruction. Therefore, if a real register has to be allocated for a virtual register of a specific sort and all real registers of this sort are in use, the real register of this sort which will not be used for the longest time will be allocated, and the value in this real register is stored in memory. So that the value of the virtual register which was previously stored in this real register can be loaded again from this memory location if this virtual register is used again later.

But, if a real register is allocated when all registers of a specific sort are in use, it is not necessary to store the register in memory if this register contains a value which has not been changed since the last time the register was loaded from memory, because the memory location where the register was loaded from contains the same value as the register, so the next time this value is used, it can still be loaded from this memory location.

The strategy described above may assign a virtual register to any real register of the same sort, but this causes problems in two cases:

- The virtual registers for which a STORE_R node exists have to be assigned to the real register of the same sort (address or data) and with the same number at the end of a basic block. (the number of register An is n, the number of register Dm is m)

- For a `MOVEM` instruction the registers to which the values are moved to or moved from have to be allocated in a specific order, i.e. data registers before address registers and registers with low numbers before registers with high numbers.

To solve this problem virtual registers for which a real register exists of the same sort and with the same number are always assigned to this real register, these registers will be called parameter registers. At the end of the basic block all values of these virtual parameter registers which are stored in memory are loaded in the real parameter registers. And a `MOVEM` instruction may only move values to or from virtual parameter registers. This adapted strategy is not optimal.

Now the next time a real register will be used again does not only depend on when the value stored in this register will be used again, but also on when the virtual register of the same sort and with the same number will be used again, because this virtual register has to be stored in this real register when it is used. So instead of allocating the real register which contains the value which will not be used for the longest time when all registers of the required sort are in use, the real register, of which the value and for which the virtual register of the same sort and with the same number will not be used for the longest time will be allocated.

6.6.2. Local register allocation algorithm.

The algorithm consists of two phases. During the first phase for every operand of the intermediate code, which uses a virtual register, the following is computed:

- The *next use instruction*. For an operand which uses a non parameter register, this is the instruction with the first operand after this operand which uses the value of this virtual register, if it exists. For an operand which uses a parameter register, this is the instruction with the first operand after this operand which uses this virtual register, if it exists.
- The *value use flag*. This flag is true if the value in the virtual register used by this operand will be used after this operand, otherwise it is false.

To compute the next use instructions the operands are divided into two classes:
- Operands which *use* a virtual register, i.e. operands which use the value of the virtual register to compute the result. For example both operands of an `ADD D1,D0` instruction.
- Operands which *define* a virtual register, i.e. operands which do not use the value of the virtual register to compute the result, but store the result (or part of the result) in this virtual register. For example the second operand of a `MOVE` to `D0` instruction.

To be able to determine which real register of a specific sort will not be used for the longest time during phase two of the algorithm, we have to be able to determine the order of the instructions with the next use, therefore the instructions are numbered. The last instruction of a basic block is numbered 2, the second last instruction is numbered 3, the third last 4, etc. By starting to number with 2, we can use 0 to indicate that no next use instruction exists and that the value of the virtual register will not be used any more, and 1 to indicate that no next use instruction exists in this basic block and that the value of the virtual register could be used after this basic block. The virtual registers which could be used after this basic block are the virtual registers for which a STORE_R node exists.

The first phase of the algorithm stores the next_use instruction and value_use flag for every operand which uses a virtual register by walking the instructions from the last instruction to the first instruction of the basic block. The instructions are walked backwards, because then the information can be computed faster. During this walk the arrays first_use and first_value_use remember for every virtual register:
- The *first use instruction*. For a non parameter register, this is the instruction with the first operand after the current operand which uses the value of this virtual register, if it exists. For a parameter register, this is the instruction with the first operand after the current operand which uses this virtual register, if it exists.
- The *first value use flag*. This flag is true if the value in the virtual register will be used after the current operand, otherwise it is false.

The first phase of the algorithm is:

```
PROCEDURE compute_next_uses
      FOR all virtual registers v DO
            IF a STORE_R node exists for v THEN
                  first_use [v] := 1
                  first_value_use [v] := TRUE
            ELSE
                  first_use [v] := 0
                  first_value_use [v] := FALSE
      instruction_number := 2
      FOR all instructions i from the last instruction to the first instruction of
                  this basic block DO
            FOR all operands p of instruction i from the last operand to the first
                        operand DO
                  IF p uses virtual register v THEN
                        next_use [p] := first_use [v]
                        value_use [p] := first_value_use [v]
                        first_use [v] := instruction_number
                        first_value_use [v] := TRUE
                  ELSE IF p defines virtual register v THEN
                        next_use [p] := first_use [v]
                        value_use [p] := first_value_use [v]
                        IF v is a parameter register THEN
                              first_use [v] := instruction_number
                        ELSE
                              first_use [v] := 0
                        first_value_use [v] := FALSE
            instruction_number := instruction_number + 1
END
```

During the second phase of the algorithm the real registers are allocated, instructions to store real registers in memory and to load real registers from memory are inserted, and the virtual registers are replaced by real registers.

To do this the operands are divided into three classes:

- Operands which *use* a virtual register, i.e. operands which use the value of the virtual register to compute the result, but don't change the value of the virtual register. For example the first operand of an `ADD D1,D0` instruction.
- Operands which *define* a virtual register, i.e. operands which do not use the value of the virtual register to compute the result, but store the result (or part of the result) in this virtual register. For example the second operand of a `MOVE` to `D0` instruction.
- Operand which *use and define* a virtual register, i.e. operands which use the value of the virtual register to compute the result, and may change the value of the virtual register. For example the second operand of an `ADD D1,D0` instruction.

The second phase of the algorithm is:

```
PROCEDURE allocate_registers
      FOR all virtual registers v DO
            real_register [v] := NO_REGISTER
      FOR all real registers r DO
            virtual_reg [r] := NO_REGISTER
            reg_changed [r] := FALSE
            real_first_use [r] := 0

      FOR all parameter registers v DO
            IF first_value_use [v] THEN
                  real_register [v] := v
                  virtual_reg [v] := v
                  real_first_use [v] := first_use [v]

      FOR all instructions i from the first instruction to the last instruction of
                  this basic block DO
            pset := all operands p of instruction i using registers
            IF pset is empty THEN
                  /* instruction i has no operands using registers, do nothing */
            ELSE IF pset contains one operand p using virtual register v THEN
                  /* get real register which has been allocated for v */
                  real_reg := real_register [v]
                  /* no real register allocated ? */
                  IF real_reg = NO_REGISTER THEN
                        /* allocate a real register */
                        IF v is a parameter register THEN
                              real_reg := v
                        ELSE
                              real_reg := real register r of the same sort as v for
                                    which real_first_use [r] is minimal

                        /* free this allocated register */
                        old_reg = virtual_reg [real_reg]
                        IF old_reg <> NO_REGISTER THEN
                              IF first_value_use [old_reg] THEN
                                    IF  reg_changed  [real_reg]  OR  old_reg  has  never
                                                been stored THEN
                                    store register old_reg
                              real_register [old_reg] := NO_REGISTER

                        virtual_reg [real_reg] := v
                        real_register [v] := real_reg

                        /* load the real register if necessary */
                        IF p uses virtual register v OR p uses and defines virtual
                                    register v THEN
                              load register real_reg
                              reg_changed [real_reg] := FALSE

                  first_use [v] := next_use [p]
                  first_value_use [v] := value_use [p]
```

```
                real_first_use [real_reg] :=
                      MAX (first_use [real_reg],first_use [v])

                IF p defines virtual register v OR p uses and defines virtual
                           register v THEN
                      reg_changed [real_reg] := TRUE

                /* replace virtual register of operand by real register */
                replace register v of operand p by real_reg
          ELSE
                similar to when pset contains one element, but more complex
                because all virtual registers used by operands in pset have to be
                in real registers at the same time
END
```

If an instruction has more than one operand which uses registers, we can not simply allocate the real registers one by one. If for example real registers have to be allocated for an `ADD D10,D3` instruction and virtual register `D10` has been allocated to register `D3` before this `ADD` instruction, then for the first operand real register `D3` would be allocated, but for the second operand `D3` would also be allocated and an instruction would be generated to store the value of virtual register `D10` from real register `D3` in memory, then an instruction would be generated to load the value of virtual register `D3` in real register `D3` from memory and then an `ADD D3,D3` instruction would be generated, which would compute `D3+D3` instead of `D10+D3`.

To prevent this, a parameter register used by an operand of the instruction or a real register which has already been allocated for a parameter register used by an operand of this instruction is never allocated for a non parameter register. Then for the example of the `ADD D10,D3` instruction, for the first operand register `D3` may not be allocated, so another register is allocated, for example `D4`, and a `MOVE D3,D4` instruction is generated, then `D3` is allocated for the second operand and an instruction is generated which loads the value of virtual register `D3` from memory into real register `D3` and then an `ADD D4,D3` instruction is generated.

Note that because for virtual parameter registers always the real register of the same sort and with the same number as the virtual register is allocated, this register allocation algorithm does not have to executed if all virtual registers used by a basic block are parameter registers, which happens very often, but only have to replace the virtual registers by the real registers of the same sort and with the same number.

6.6.3. Preserving condition codes during local register allocation.

Many load, store and register to register move instructions inserted in the code during local register allocation change the condition codes of the MC68020, so that sometimes the code will not be correct. Therefore different instructions are generated for loads, stores and register to register moves if the condition codes may not be altered:

- For a load a `MOVE` or `MOVEA` instruction is generated. If the destination of the move instruction is an address register (`MOVEA`), the condition codes are not affected, so we don't have to use an other instruction. But if the destination of the move instructions is a data register (`MOVE`), the condition codes are effected. Therefore a `MOVEM` to this data register is generated instead, which is slower than a `MOVE`, but doesn't affect the condition codes.

- For a store a `MOVE` instruction is generated, which always affects the condition codes. Therefore a `MOVEM` instruction is used instead, which is slower than a `MOVE`, but doesn't affect the condition codes.

- For a register to register move a `MOVE` or `MOVEA` instruction is generated. If the destination of the move is an address register (`MOVEA`), the condition codes are not affected, so we don't have to use an other instruction. But if the destination is a data register (`MOVE`), the condition codes are affected. Therefore an `EXG` instruction is generated instead, which doesn't affect the condition codes.

6.7. Optimizing stack accesses.

We have represented accesses to the stack using LOAD, FLOAD, STORE and FSTORE nodes in the dag. These nodes represent an element on a stack by a displacement and a stackpointer, so that the address of the stack element is the sum of this displacement and the address in this stackpointer at the beginning of the basic block. And from these nodes we have generated code which accesses the stack elements using an address register indirect with displacement addressing mode. (also called indirect with displacement addressing mode for short)

But the MC68020 has two addressing modes to push and pop elements on and off a stack, i.e. the address register indirect with postincrement and address register indirect with predecrement addressing modes. Addressing stack elements using these addressing modes is usually more efficient than addressing stack elements using the address register indirect with displacement addressing mode, because:

- Using an indirect with displacement addressing mode instead of a postincrement or predecrement addressing mode makes the machine code of an instruction one word longer, because the displacement is one word long and is part of the instruction. (see section 2.3.7 or appendix B)

- The indirect with displacement addressing mode is usually a bit slower than the predecrement addressing mode and as fast as the postincrement addressing mode, when executing from the cache. But if the instruction is not in the cache, the address register with displacement addressing mode will usually be slower, because the machine code of the instruction is longer, and so it may take more time to fetch the instruction from memory.

- If the indirect with displacement addressing mode is used, usually a constant has to be added to a stackpointer at the end of the basic block to adjust it, but if the postincrement and predecrement addressing modes are used, this is often not necessary.

If the displacement of the indirect with displacement addressing mode is zero, the address register indirect addressing mode can better be used, because this addressing mode is usually faster and uses less memory.

All A and B-stack elements are accessed using an address register indirect with displacement addressing mode, where the address register is the A or B-stackpointer. The A and B-stackpointers are never changed in a basic block, but only at the end eventually a constant is added to them. This makes optimizing these accesses to address register indirect, postincrement and predecrement addressing modes very easy.

For all A and B-stack accesses, except the last A or B-stack access of a basic block, the following optimizations are performed:

- If the displacement of the stack access is zero and the size of the element which is accessed by this stack access is equal to the displacement of the next stack access, then the addressing mode of this stack access is changed to a postincrement addressing mode. Then for the next stack access an indirect or a postincrement addressing mode can be used.

- If the displacement of the stack access is zero and the size of the element which is accessed by this stack access - the size of the element which is accessed by the next stack access is equal to the displacement of the next stack access and not zero, then the addressing mode is changed to a postincrement addressing mode. Then for the next stack access a predecrement addressing mode can be used.

- If the displacement of the stack access is equal to 0 - the size of the element which is accessed by this stack access and the displacement of the next stack access is not zero, then the addressing mode of this stack access is changed to a predecrement addressing mode.

And for the last A and B-stack accesses of a basic block the following optimizations are performed:

- If the displacement of the stack access is zero and the size of the element which is accessed by this stack access is equal to the constant which has to be added to the stackpointer at the end of the basic block, then the addressing mode of this stack access is changed to a postincrement addressing mode.

- If the displacement of the stack access is equal to 0 - the size of the element which is accessed by this stack access and the constant which has to be added to the stackpointer at the end of the basic block is not zero, then the addressing mode of this stack access is changed to a predecrement addressing mode.

If the addressing mode of a stack access is changed to a postincrement addressing mode, the size of the element which is accessed by this stack access has to be subtracted from the displacements of all the following stack accesses and from the constant which has to be added to the stackpointer at the end of the basic block. And if the addressing mode of a stack access is changed to a predecrement addressing mode, the size of the element which is accessed by this stack access has to be added to the displacements of all the following stack accesses and to the constant which has to be added to the stackpointer at the end of the basic block.

If the constant, which has to be added to a stackpointer at the end of the basic block is zero, then of course no add instruction is necessary to adjust the stackpointer. The stack accesses using an address register indirect with displacement addressing mode for which the displacement is zero are optimized to an address register indirect addressing mode (without displacement) when generating MC68020 machine code from the intermediate code. (see section 6.10)

Examples of stack access optimizations can be found in appendix A, for example A.1.6 and A.1.7.

The stack access optimizations described above only look one stack access ahead, by looking more stack accesses ahead sometimes better results can be obtained. For example the basic block: (all the examples below are of intermediate code, except that address register indirect with displacement addressing modes for which the displacement is zero are optimized to address register indirect addressing modes)

```
MOVE        (BSP),D0
ADD         4(BSP),D0
MOVE        D0,-4(BSP)
SUB         #4,BSP
```

would be optimized to:

```
MOVE        (BSP)+,D0
MOVE        (BSP),D0
MOVE        D0,-8(BSP)
SUB         #8,BSP
```

but could better have been optimized to:

```
MOVE        (BSP),D0
MOVE        4(BSP),D0
MOVE        D0,-(BSP)
```

The constant which has to be added to a stackpointer is currently always added (or subtracted) at the end of the basic block, but sometimes it is more efficient to do this sooner. For example the basic block:

```
MOVE        8(BSP),D0
ADD         #8,BSP
```

could be optimized, by moving the addition to the stackpointer, to:

```
ADD         #8,BSP
MOVE        (BSP),D0
```

Also the stack accesses can sometimes be made more efficient by changing the order of a few instructions. For example the basic block:

```
MOVE        4(BSP),D0
MOVE        (BSP),D1
ADD         #8,BSP
```

would be optimized to:

```
MOVE        4(BSP),D0
MOVE        (BSP),D1
ADDQ.L      #8,BSP
```

but could better have been optimized, by changing the order of the first two instructions, to:

```
MOVE.L      (BSP)+,D1
MOVE.L      (BSP)+,D0
```

6.8. Optimizing jumps.

To optimize branches and jumps as described in section 5.1.2, the code generator searches for three successive basic blocks A, B and C, for which the following conditions are true:

- Basic block B consist of just one `JMP` instruction to a label within this module.
- Basic block B is not labeled by a label.
- Basic block A has as last instruction a branch instruction (`BEQ`, `BGE`, `BGT`, `BLE`, `BLT`, `BNE`, `FBEQ`, `FBGE`, `FBGT`, `FBLE`, `FBLT` or `FBNE`) which branches to a label which labels basic block C.

If such a basic block is found, the branch instruction is changed to a branch instruction with the reverse condition code (i.e. `EQ` -> `NE`, `GE` -> `LT`, `GT` -> `LE`, `LE` -> `GT`, `LT` -> `GE` and `NE` -> `EQ`) and with as operand the label which is the operand of the `JMP` instruction, and the `JMP` instruction is removed.

For example: (basic block 6,7 and 8 of appendix A.1.7 and A.1.8)

Basic block A:

```
sFac.1:     CMP         #0,D0
            BEQ         m.2
```

Basic block B:

```
            JMP         sFac.2
```

Basic block C:

```
m.2:        MOVE        #1,D0
            RTS
```

is optimized to:

Basic block A:

```
sFac.1:     CMP         #0,D0
            BNE         sFac.2
```

Basic block B:

Basic block C:

```
m.2:        MOVE        #1,D0
            RTS
```

6.9. Calling the garbage collector.

If space has to be allocated in the heap for new nodes or arguments, this can be done by subtracting the number of long words, which have to be allocated, from FC and calling the garbage collector if FC becomes less than zero. But in section 5.1.1 we saw this can be done faster by allocating space in the heap for more nodes at once. This has been implemented by allocating space in the heap for all nodes which are created of filled in the basic block at once at the beginning of the basic block.

The number of long words which is allocated in the heap by a create or fill instruction is a constant. So the number of long words which are allocated by all the create and fill instructions of a basic block is a constant as well.

But the `add_args` instruction allocates a number of long words which is not a constant. Therefore we allocate the maximum number of long words which an `add_args` instruction would allocate. Then the number of long words to be allocated by a basic block is always a constant.

This number of long words to be allocated can be computed by calculating the sum of the number of long words which the CREATE and ALLOCATE nodes in the dag of this basic block allocate.

Then for every basic block for which the number of long words is not zero, the following code is generated at the start of the basic block:

```
       SUB.L     #number_of_long_words,FC
       BMI       jmp_gc_1
continue_from_gc_1:

and at the end of this module:

jump_gc_1:
       JSR       collect_n
       BRA       continue_from_gc_1
```

If the number of long words to be subtracted from `FC` is less than 8, a `SUBQ.L` instruction is used. Otherwise if the number of long words to be allocated is less than 128 and a data register is available, the number of long words is first moved to a free data register Dn using `MOVEQ` and then subtracted using a `SUB Dn,FC`. Otherwise a `SUB.L #number_of_long_words,FC` instruction is used.

The garbage collector has to be able to find all pointers to nodes in the graph including the ones in address registers. Because after global register allocation we know which address registers contain elements of the A-stack, and because all elements of the A-stack contain pointers to nodes, we know that the address registers which contain A-stack elements are the only registers containing pointers to nodes. At the start of a basic block there are only four possible address register allocations: no address registers used, A0 used, A0 and A1 used, and A0, A1 and A2 used. By making an entry in the garbage collector for each of these four allocations, the garbage collector can be called by just a `JSR` instruction.

Maybe you wonder why the garbage collector is not called by:

```
       SUB.L     #number_of_long_words,FC
       BPL.S     continue_from_gc_1
       JSR       collect_n
continue_from_gc_1:
```

The reason is that this instruction sequence is longer than the part of the other instruction sequence which has to be stored at the start of a basic block. This would probably result in a less efficient use of the instruction cache of the MC68020, because the `JSR collect_n` instruction would sometimes be stored in the cache, but will usually not be executed.

6.10. Generating MC68020 code from the intermediate code.

After local register allocation (and also after optimizing stack accesses) the intermediate code is very similar to MC68020 code. Nearly all instructions of the intermediate code can be translated directly to an MC68020 instruction, except:

- The `SEQ`, `SGE`, `SGT`, `SLE`, `SLT`, `SNE`, `FSEQ`, `FSGE`, `FSGT`, `FSLE`, `FSLT` and `FSNE` compute a long word, but the Scc instructions of the MC68020 compute a byte, therefore an extra `EXTB.L` instruction is necessary to sign-extend the byte to a long word.

- The `BMOVE` instruction moves a block of long words. But the MC68020 doesn't have a block move instruction, therefore the block move is performed using a loop. For example a:

```
BMOVE     (A0)+,(A1)+,D0
```

  instruction is translated to:

```
      BRA.S     label_1
label_2:
      MOVE.L    (A0)+,(A1)+
label_1:
      DBRA      D0,label_2
```

While generating this code, the following simple optimizations are performed: (most of these optimizations can be performed by assemblers, but because object code (see appendix E) is generated by this code generator, the code generator has to do these optimizations)

- An address register indirect with displacement addressing mode of which the displacement is zero is optimized to an address register indirect addressing mode. (without displacement).
- `MOVE.L #i,Dn` for which $-128 \leq i \leq 127$ is optimized to `MOVEQ #i,Dn`.
- `ADD.L #i,Rn` (`Rn` is Dn or An) for which $1 \leq i \leq 8$ is optimized to `ADDQ.L #i,Rn`.
- `ADD.L #i,Rn` for which $-8 \leq i \leq -1$ is optimized to `SUBQ.L #-i,Rn`.
- `ADD.L #i,An` for which $-32768 \leq i \leq 32767$ is optimized to `LEA i(An),An`. (except for $-8 \leq i \leq -1$ or $1 \leq i \leq 8$)
- `SUB.L #i,Rn` for which $1 \leq i \leq 8$ is optimized to `SUBQ.L #i,Rn`.
- `SUB.L #i,Rn` for which $-8 \leq i \leq -1$ is optimized to `ADDQ.L #-i,Rn`.
- `SUB.L #i,An` for which $-32767 \leq i \leq 32768$ is optimized to `LEA -i(An),An`. (except for $-8 \leq i \leq -1$ or $1 \leq i \leq 8$)
- `CMP.L #0,<ea>` or `CMP.W #0,<ea>` is optimized to `TST.L <ea>` or `TST.W <ea>`, except if `<ea>` is address register direct.
- `MOVE.L #0,<ea>` is optimized to `CLR.L <ea>`, except if `<ea>` is data register direct or address register direct.
- `JMP label` for which the `label` is in the same module as the `JMP` is optimized to `BRA label`.
- `JSR label` for which the `label` is in the same module as the `JSR` is optimized to `BSR label`.

7. Evaluation.

In this chapter I will first describe which optimizations have been implemented and how the code generator can be improved further. Then this code generator is compared to the previous code generator, to determine by how much the generated code has been improved by the implemented optimizations.

7.1. Implemented optimizations.

By generating code as described in section 6 the following optimizations have been implemented:

- Use of registers within basic blocks is usually good, but can be improved further. (see 7.2)
- Parameters and results of functions are passed in registers. (see 5.2.2)
- The evaluation order determined by the labeling algorithm results in better use of registers, because fewer registers are used, and making fewer copies. (see 5.2.5)
- The `jsr_eval` optimization described in section 5.2.6 has been implemented.
- Nearly all `create`s of nodes are changed into a create and fill at once (see 5.1.1). The most important exception is a `create` in the following situation.
  If the result of a function is a node, the code generated by the Clean compiler for this function returns this node by overwriting a node. If such a function is called, and there is no node which can be overwritten, a node is created. Such creates are never optimized to a create and fill at once by the code generator.
- Unnecessary copies and stack manipulations are eliminated within basic blocks. (see 5.2.3) These are automatically eliminated by constructing a dag for a basic block.
- Optimization of booleans as described in section 5.2.3 is always performed.
- Jump optimization as described in section 5.1.2 is always performed.
- All MC68020 specific code optimizations described in 5.3 are performed.
- All the space which has to be allocated in the heap for a basic block is allocated at once (see 5.1.1).

7.2. Possible improvements.

The code generated by the code generator can be further improved in the following ways:

- If a value is computed by a basic block, and this value has to be stored in a register at the end of the basic block, the value is not always computed in this register by the code generated by this code generator. But the value is computed in an other register, so that an extra move instruction is necessary. By computing the value in the required register, this move can be removed.
- Allocating floating point registers of the MC68881 coprocessor at the beginnings and ends of basic blocks for floating point numbers. And passing floating point arguments and results of functions in these registers.
- The code for a `fill_a` instruction can be improved, by not first loading the fixed size part of the node to be copied in registers.

- Performing in-line code substitution for small functions (see 5.1.8). But this can better be done by the Clean compiler.

- Improving global register allocation.

- Removing unused stack elements from the stack, if such a value is stored in a register and a function is called using `jsr`, so that it doesn't have to be saved before the function call. (see 5.19) Often this save is necessary because an unnecessary copy is made. (see 5.2.3)

- Extending the local register allocator, so that it can save address registers in free data registers instead of in memory if enough data registers are available, and can save data registers in address registers.

- Implementing strength reductions (see 5.1.3).

- Implementing constant folding (see 5.1.4).

- Improving the optimizations of stack accesses. (see 6.7)

- Implementing algebraic optimizations. (see 5.1.5)

- Implementing common subexpression elimination. (see 5.1.6)

- Removing unused code. (see 5.1.7)

Other possible improvements are:

- Making the representation of nodes in the heap smaller. For example by combining the evaluation address and the descriptor so that these can be stored in one long word (and not two). Or by using special representations for integers and lists. Then nodes can be created faster, and the garbage collector has to be called less often.

- Implementing a garbage collector which doesn't need half the heap as workspace. So that we can use the whole heap to store nodes, and not only half of it.

7.3. Comparing this code generator with the previous code generator.

Before I started to design and implement this code generator, a code generator [Weijers 1990] which translated ABC code to MC68020 code already existed. But this compiler generated code for the Sun instead of for the Mac II, and the code generated by this compiler was often not very efficient. This compiler generated for every instruction ABC instruction a fixed sequence of MC68020 instructions (a kind of macro substitution), but with the following optimizations:

- The top element of the B-stack is often stored in a data register or, if the top element is a boolean, in the condition codes of the MC68020.

- The code generator could detect some cases, but not as many as my code generator, in which it is not necessary to initialize a node when a node is created, because it would be filled before a garbage collection could occur. For example, for the instruction sequence: `create`, `fillI +1 0` the node is not initialized by the `create` instruction.

- Conditional jumps followed by a (unconditional) jump (e.g. `jmp_false label_1`, `jmp label_2`, `label_1:`) are optimized to one conditional jump, just like my compiler does.

To determine how much more efficient my code generator is, I have compared it to this 'old' code generator for the Sun. To do this, I have used some of the benchmarks and execution times of these benchmarks of the 'old' code generator on the Sun described in [Heerink 1990].

To determine how much faster the Sun is than the Mac II, the code generated by my code generator for the nfib, tak and r_plus benchmarks was executed on both the Sun and the Mac II. The nfib benchmark was executed 1.67 times as fast (execution time on the Mac II / execution time on the Sun) on the Sun, the tak benchmark 1.63 times as fast and the r_plus benchmark only 1.09 times as fast. I also executed a simple loop, which was executed 1.56 times as fast, and a loop with two `MOVE.L (A0),(A1)` instructions, which was executed 1.92 times as fast.

To compare my code generator to the old code generator, I have assumed the Sun is 1.65 times as fast for programs not using reals. The results for nfib and tak will then be accurate, but of course the results obtained in this way for the other benchmarks are not accurate, because the benchmarks were executed between 1.56 and 1.92 times as fast.

Because on both machines computations on reals are not performed by the MC68020, but by the MC68881 coprocessor, I have assumed the Sun is 1.09 times as fast for programs using reals, (r_plus and r_verm) because the r_plus benchmark was executed only 1.09 times as fast. The results for r_plus will then be accurate, but the results obtained in this way for r_verm will not be accurate.

In the following table the execution times of the benchmarks (first two columns), the execution time of the old code on the Mac II (third column), and the speed up factor (last column) are given for the benchmarks. (assuming the Sun is 1.65 or 1.09 times as fast) The execution times are including the time for garbage collections and for a heap size of two mega bytes. (two semi-spaces of one mega byte)

benchmark:	execution time (s) Mac II my code	execution time (s) Sun 'old' code	execution time (s) Sun * (1.65 | 1.09) ' old' code	speed up factor 'old' code / new code
nfib	0.81	0.66	1.1	1.3
tak	0.22	0.26	0.4	1.9
i_plus	0.39	0.68	1.1	2.9
i_verm	1.79	1.36	2.2	1.3
r_plus	3.02	5.42	5.9	2.0
r_verm	2.74	4.17	4.6	1.7
spine	2.19	2.70	4.5	2.0
twice	2.21	2.46	4.1	1.8
match	12.8	9.89	16	1.3
reverse	10.5	9.38	16	1.5
nfiblist	2.43	1.66	2.7	1.1

From these results we can conclude:

- Computations on strict basic values are a lot faster. Specially for simple operations like adding and subtracting, because i_plus (integer additions and subtractions) is executed about 2.9 times as fast and r_plus (real additions and subtractions) is executed 1.9 times as fast. For more complex operations like multiplying and dividing the improvement is smaller, but still i_verm (integer multiplications and divisions) is about 1.3 times as fast and r_verm (real additions and subtractions) is about 1.7 times as fast. The reasons for the faster execution of these benchmarks are that values are kept in registers during the computations and integer parameters are passed in registers.

- Parameter passing of strict arguments is faster, because tak is 1.9 times as fast, and nfib is 1.3 times as fast. This is largely because now parameters are passed and stored in registers.

- Curried function applications are executed faster, because spine is executed 2.0 times as fast and twice is executed 1.8 times as fast. But this is largely due to writing the code for AP in assembly, instead of generating this code with the code generator, and by using a better parameter passing convention for apply entries of functions. But also because of the `jsr_eval` optimization described in section 5.2.6 and passing parameters in registers.

- Functions which manipulate lists are a bit faster, because reverse is about 1.5 times as fast and nfiblist is about 1.1 times as fast. Reverse is faster largely due to the `jsr_eval` optimization described in section 5.2.6 and better use of registers. Nfiblist is not much faster, because many nodes are created and filled by this benchmark.

- Pattern matching on non strict arguments is faster, because match is executed about 1.3 times as fast. This is because parameters are passed in registers and the `jsr_eval` optimization described in section 5.2.6.

So, briefly we can conclude that computations on strict arguments and basic values (integers, reals, etc.) are done a lot faster (about 1.3 - 2.9 times as fast) by the code generated by my code generator, but that computations on non strict arguments are only a bit faster (about 1.1 - 1.5 times as fast). Curried function applications are also executed a lot faster (about 1.8 - 2.0 times as fast), but not so much due to better code generation techniques.

APPENDIX A: Examples.

A.1. Fac.

A.1.1. Fac in Clean.

```
Fac 0       ->    1                  |
Fac n       ->    *I n (Fac (--I n))      ;
```

A.1.2. ABC code of fac.

```
      .o            3 0
lFac:                                 || apply entry : 'Fac' node, argument n
                                         node and node to be overwritten by
                                         result on the A-stack
      pop_a         1                 || pop 'Fac' node from the A-stack
      jmp           m.1               || jump to label m1
      .o            1 0
nFac:                                 || node entry : 'Fac n' node on the A-
                                         stack
      push_args     0 1 1             || push argument n on the A-stack
m.1:
      set_entry     _cycle_in_spine 1 || store _cycle_in_spine as evaluation
                                         address to detect cycles in the spine
                                         of the graph
      jsr_eval                        || evaluate argument n
      pushI_a       0                 || push argument n on the B-stack
      pop_a         1                 || pop node of argument n from the A-stack
      .d            0 1 i
      jsr           sFac.1            || call strict entry of Fac to compute
                                         result on the B-stack
      .o            0 1 i
      fillI_b       0 0               || fill the result node with the integer
                                         result on the B-stack
      pop_b         1                 || pop the integer result from the B-stack
      .d            1 0
      rtn                             || return
      .o            0 1 i
sFac.1:                               || strict entry : argument n (evaluated)
                                         on the B-stack
      eqI_b         +0 0              || n equal 0 ?
      jmp_true      m.2               || yes, jump to label m.2
      jmp           sFac.2            || no, jump to label sFac.2
m.2:
      pop_b         1                 || pop argument n from the B-stack
      pushI         +1                || push 1 (result) on the B-stack
      .d            0 1 i
      rtn                             || return
sFac.2:
      push_b        0                 || push argument n on the B-stack
      decI                            || subtract 1 from copy of n on the B-
                                         stack
      .d            0 1 i
      jsr           sFac.1            || call strict entry of fac with argument
                                         n-1 on the B-stack to compute Fac (--I
                                         n)
      .o            0 1 i
      push_b        1                 || push argument n on the B-stack
```

```
update_b        1 2                     || reorganize the B-stack
update_b        0 1
pop_b           1
mulI                                    || compute *I n (Fac (--I n))
.d              0 1 i
rtn                                     || return
```

A.1.3. Basic blocks of the ABC code of fac.

Basic block 1:

```
                .o              3 0
        lFac:   pop_a           1
                jmp             m.1
```

Basic block 2:

```
                .o              1 0
        nFac:   push_args       0 1 1
```

Basic block 3:

```
        m.1:    set_entry       _cycle_in_spine 1
                jsr_eval
```

Basic block 4:

```
                pushI_a         0
                pop_a           1
                .d              0 1 i
                jsr             sFac.1
```

Basic block 5:

```
                .o              0 1 i
                fillI_b         0 0
                pop_b           1
                .d              1 0
                rtn
```

Basic block 6:

```
                .o              0 1 i
        sFac.1: eqI_b           +0 0
                jmp_true        m.2
```

Basic block 7:

```
                jmp             sFac.2
```

Basic block 8:

```
        m.2:    pop_b           1
                pushI           +1
                .d              0 1 i
                rtn
```

Basic block 9:

```
        sFac.2:
                push_b          0
                decI
                .d              0 1 i
                jsr             sFac.1
```

Basic block 10:

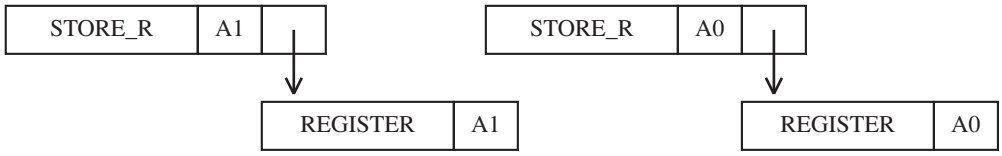
```
                .o              0 1 i
                push_b          1
                update_b        1 2
                update_b        0 1
                pop_b           1
```

```
mulI
.d                0 1 i
rtn
```

A.1.4. Dag representation and global register allocation of fac.

Basic block 1:

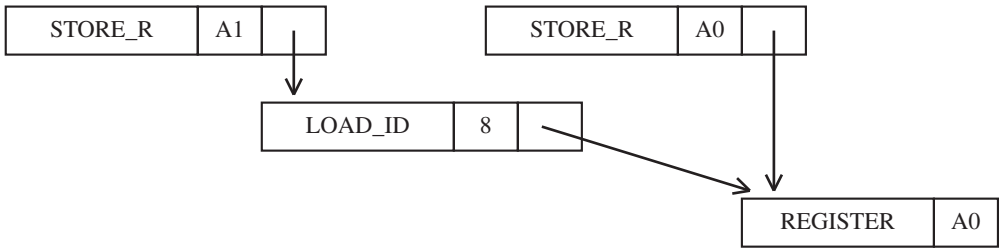
```
lFac:                       || registers A0-A2 allocated
```




```
                            || registers A0-A1 allocated
jmp               m.1
```

Basic block 2:

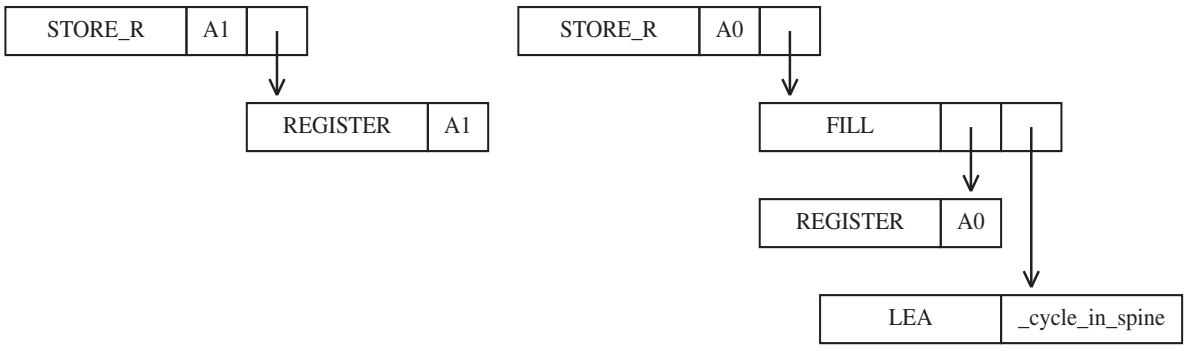
```
nFac:                       || register A0 allocated
```




```
                            || registers A0-A1 allocated
```

Basic block 3:

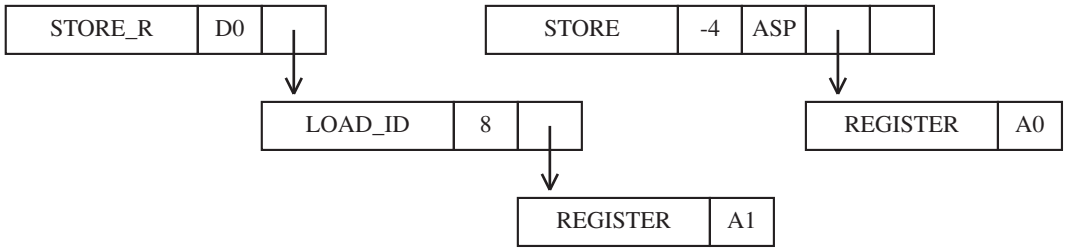
```
m.1:                        || registers A0-A1 allocated
```




```
                            || registers A0-A1 allocated
jsr_eval
```

Basic block 4:

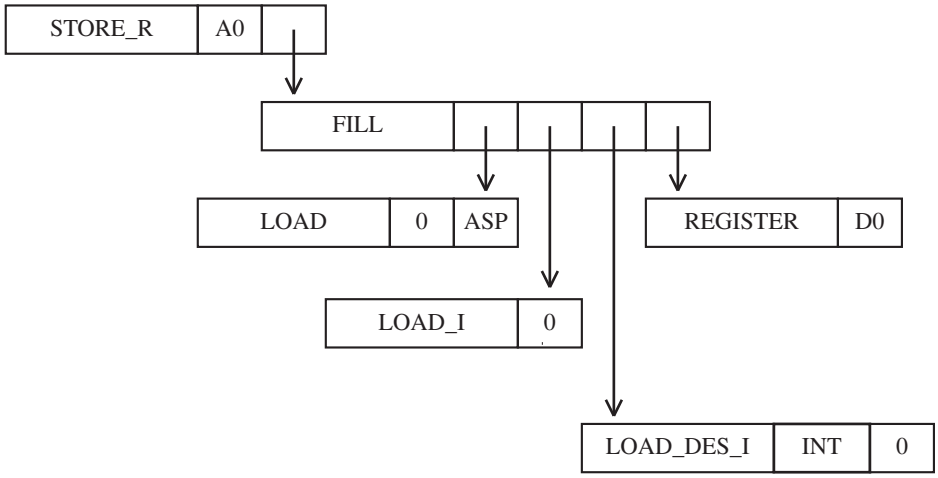
```
                            || registers A0-A1 allocated
```



```
                                || register D0 allocated, add -4 to A-
                                   stackpointer at end of basic block
        jsr                     sFac.1
```

Basic block 5:

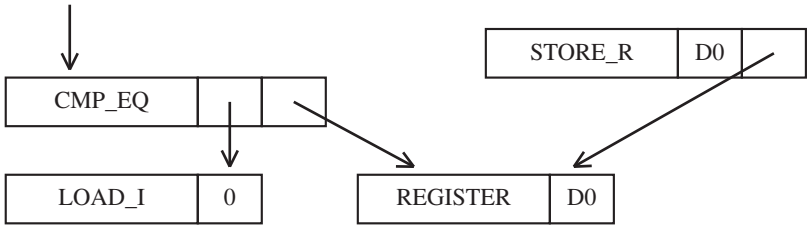
```
                                || register D0 allocated
```




```
                                || register A0 allocated, add 4 to A-
                                   stackpointer at end of basic block
        rtn
```

Basic block 6:

```
sFac.1:                         || register D0 allocated
```



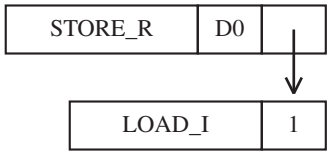

```
                                || register D0 allocated
        jmp_true                m.2
```

Basic block 7:

```
                                || register D0 allocated
        jmp                     sFac.2
```

Basic block 8:

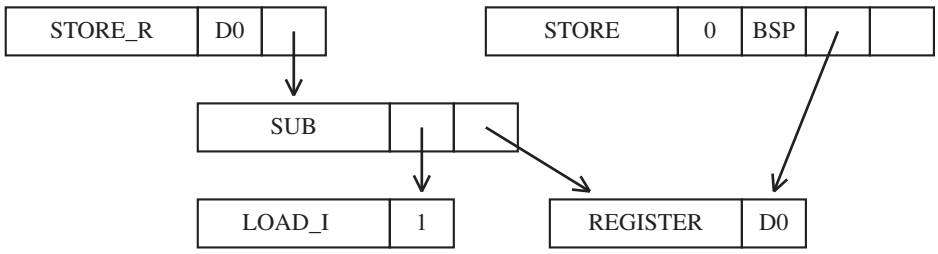
```
m.2:                            || register D0 allocated
```




```
                                || register D0 allocated
        rtn
```

Basic block 9:

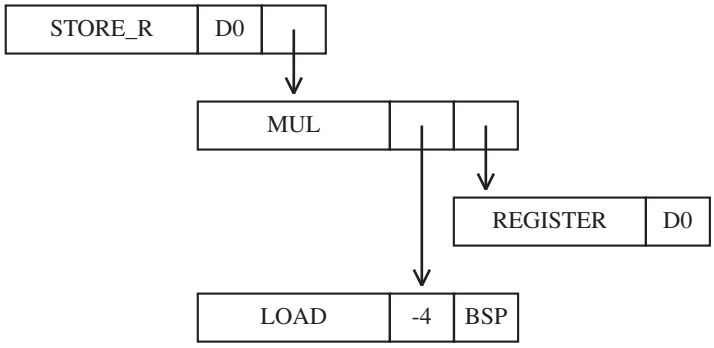
```
sFac.2:                         || register D0 allocated
```




```
                                || register   D0   allocated,   add   4   to   B-
                                   stackpointer at end of basic block
                jsr             sFac.1
```

Basic block 10:

```
                                || register D0 allocated
```




```
                                || register   D0   allocated,   add   -4   to   B-
                                   stackpointer at end of basic block
                rtn
```

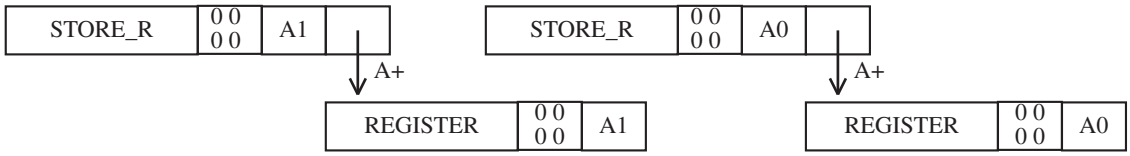
A.1.5. Dag representation of fac after computing the increases and uses of the number of registers and global register allocation.

In the boxes to the right of the box containing the name of the node 4 numbers are given:
- The number above left is the (additional) number of address registers which are used when the cdag with as root this node is evaluated.
- The number above right is the (additional) number of data registers which are used when the cdag with as root this node is evaluated.
- The number below left is the increase in the number of used address registers when the cdag with as root this node is evaluated.
- The number below right is the increase in the number of used data registers when the cdag with as root this node is evaluated.

The result of a node can be evaluated in a data register, address register or in memory. Evaluating in memory is evaluating so that the value can be accessed using an address register indirect, postincrement, predecrement or immediate addressing mode. In the graphs below this is indicated by a D (data register), A (address register) or M (memory) near the arrow. Such a D, A or M can be followed by a + to indicate that the register may be released after the value has been accessed. For an M+ this means that an address register may be released after the value has been accessed. This may happen if the memory location is accessed with an address register indirect, postincrement or predecrement addressing mode.

Basic block 1:

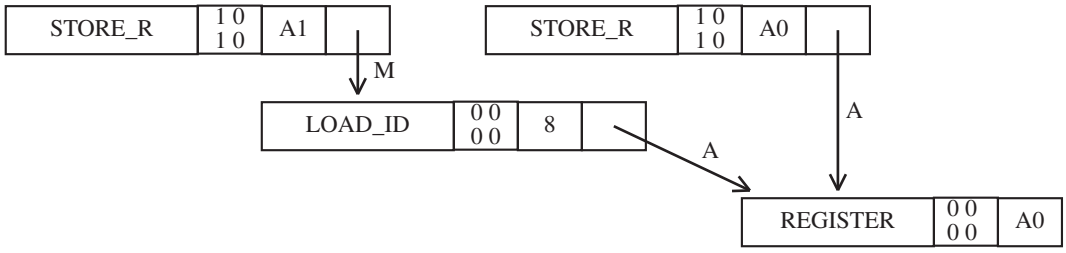
```
                lFac:           || registers A0-A2 allocated
```




```
                        || registers A0-A1 allocated
        jmp             m.1
```

Basic block 2:

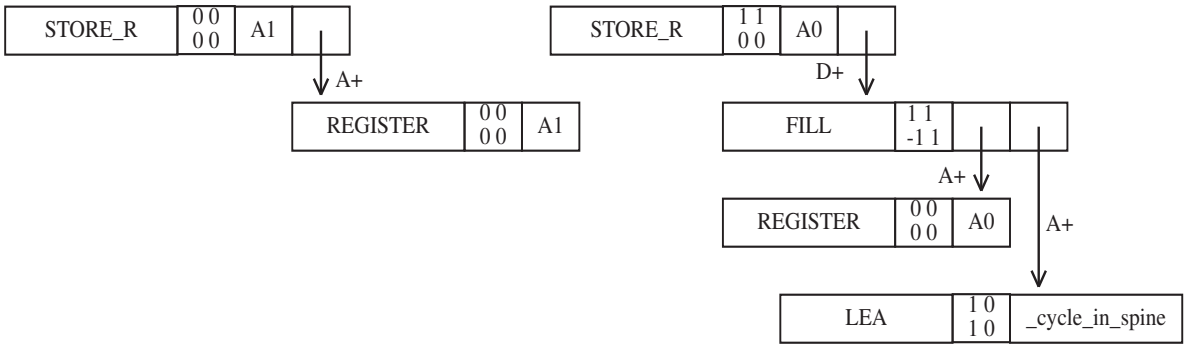
```
nFac:                   || register A0 allocated
```




```
                        || registers A0-A1 allocated
```

Basic block 3:

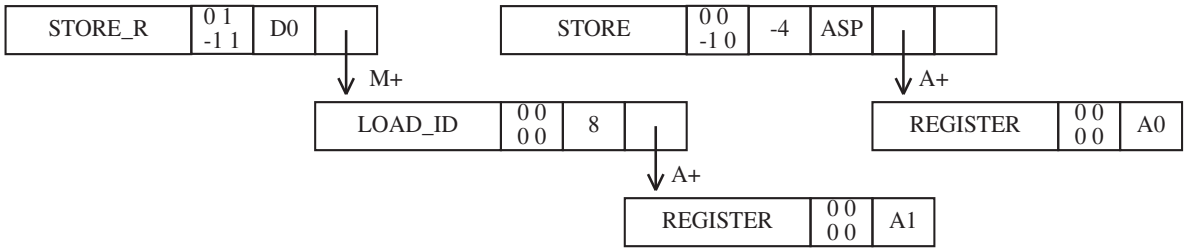
```
m.1:                    || registers A0-A1 allocated
```




```
                        || registers A0-A1 allocated
        jsr_eval
```

Basic block 4:

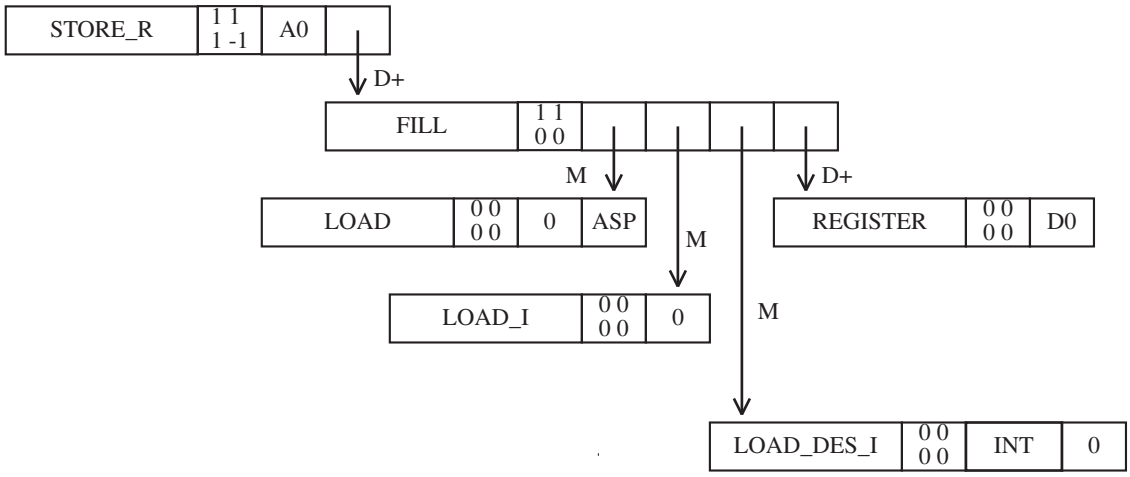
```
                        || registers A0-A1 allocated
```




```
                        || register  D0  allocated,  add  -4  to  A-
                           stackpointer at end of basic block
        jsr             sFac.1
```

Basic block 5:

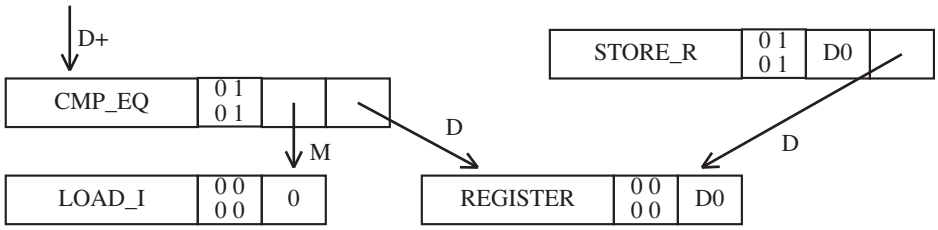
```
                        || register D0 allocated
```




```
                                || register  A0  allocated,  add  4  to  A-
                                   stackpointer at end of basic block
        rtn
```

Basic block 6:

```
sFac.1:                         || register D0 allocated
```



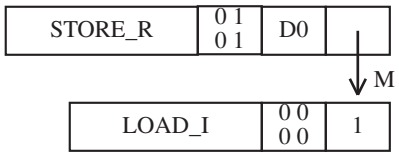

```
                                || register D0 allocated
        jmp_true                m.2
```

Basic block 7:

```
                                || register D0 allocated
        jmp                     sFac.2
```

Basic block 8:

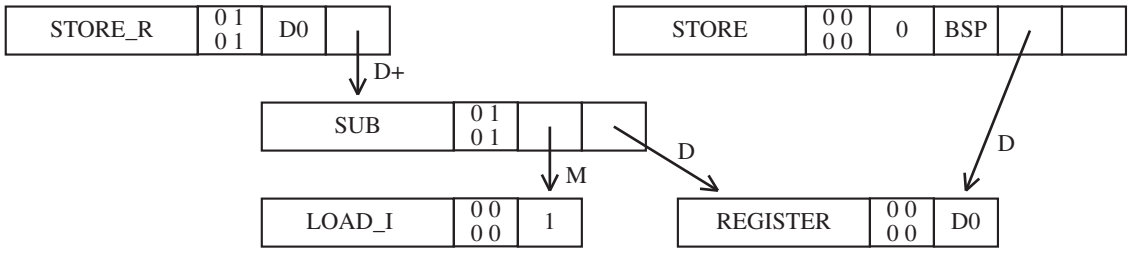
```
m.2:                            || register D0 allocated
```




```
                                || register D0 allocated
        rtn
```

Basic block 9:

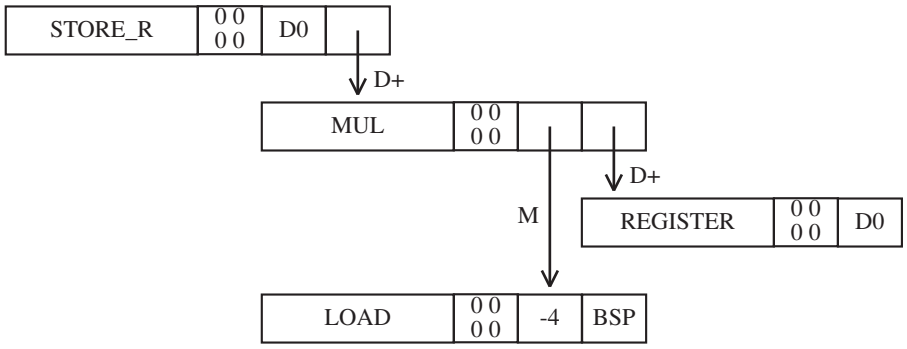
```
sFac.2:                         || register D0 allocated
```



```
                        || register  D0  allocated,  add  4  to  B-
                           stackpointer at end of basic block
            jsr         sFac.1
```

Basic block 10:

```
                        || register D0 allocated
```




```
                        || register  D0  allocated,  add  -4  to  B-
                           stackpointer at end of basic block
            rtn
```

A.1.6. Intermediate code of fac generated from the dag.

Basic block 1:

```
            lFac:       JMP         m.1
```

Basic block 2:

```
            nFac:       MOVE        8(A0),A1
                        MOVE        0(A1),A1
```

Basic block 3:

```
            m.1:        LEA         _cycle_in_spine,A2
                        MOVE        A2,0(A0)
                        MOVE        0(A1),D6
                        BEQ         eval_0
                        MOVE        A0,-(A3)
                        MOVE        A1,A0
                        MOVE        D6,A1
                        JSR         0(A1)
                        MOVE        A0,A1
                        MOVE        (A3)+,A0
```

Basic block 4:

```
            eval_0:     MOVE        A0,-4(A3)
                        MOVE        8(A1),D0
                        ADD         #-4,A3
                        JSR         sFac.1
```

Basic block 5:

```
                        MOVE        0(A3),A0
                        MOVE        A0,D1
                        MOVE        #0,(A0)+
                        MOVE        #!INT+0,(A0)+
                        MOVE        D0,0(A0)
                        MOVE        D1,A0
                        ADD         #4,A3
                        RTS
```

Basic block 6:

```
sFac.1:     CMP         #0,D0
            BEQ         m.2
```

Basic block 7:

```
            JMP         sFac.2
```

Basic block 8:

```
m.2:        MOVE        #1,D0
            RTS
```

Basic block 9:

```
sFac.2:     MOVE        D0,0(A4)
            SUB         #1,D0
            ADD         #4,A4
            JSR         sFac.1
```

Basic block 10:

```
            MUL         -4(A4),D0
            SUB         #4,A4
            RTS
```

A.1.7. Intermediate code of fac after stack access optimization.

Basic block 1:

```
lFac:       JMP         m.1
```

Basic block 2:

```
nFac:       MOVE        8(A0),A1
            MOVE        0(A1),A1
```

Basic block 3:

```
m.1:        LEA         _cycle_in_spine,A2
            MOVE        A2,0(A0)
            MOVE        0(A1),D6
            BEQ         eval_0
            MOVE        A0,-(A3)
            MOVE        A1,A0
            MOVE        D6,A1
            JSR         0(A1)
            MOVE        A0,A1
            MOVE        (A3)+,A0
```

Basic block 4:

```
eval_0:     MOVE        A0,-(A3)
            MOVE        8(A1),D0
            JSR         sFac.1
```

Basic block 5:

```
            MOVE        (A3)+,A0
            MOVE        A0,D1
            MOVE        #0,(A0)+
            MOVE        #!INT+0,(A0)+
            MOVE        D0,0(A0)
            MOVE        D1,A0
            RTS
```

Basic block 6:

```
sFac.1:     CMP         #0,D0
            BEQ         m.2
```

Basic block 7:

```
                JMP         sFac.2
```

Basic block 8:

```
        m.2:    MOVE        #1,D0
                RTS
```

Basic block 9:

```
        sFac.2: MOVE        D0,(A4)+
                SUB         #1,D0
                JSR         sFac.1
```

Basic block 10:

```
                MUL         -(A4),D0
                RTS
```

A.1.8. Intermediate code of fac after stack access optimization and jump optimization.

Basic block 1:

```
        lFac:   JMP         m.1
```

Basic block 2:

```
        nFac:   MOVE        8(A0),A1
                MOVE        0(A1),A1
```

Basic block 3:

```
        m.1:    LEA         _cycle_in_spine,A2
                MOVE        A2,0(A0)
                MOVE        0(A1),D6
                BEQ         eval_0
                MOVE        A0,-(A3)
                MOVE        A1,A0
                MOVE        D6,A1
                JSR         0(A1)
                MOVE        A0,A1
                MOVE        (A3)+,A0
```

Basic block 4:

```
        eval_0: MOVE        A0,-(A3)
                MOVE        8(A1),D0
                JSR         sFac.1
```

Basic block 5:

```
                MOVE        (A3)+,A0
                MOVE        A0,D1
                MOVE        #0,(A0)+
                MOVE        #!INT+0,(A0)+
                MOVE        D0,0(A0)
                MOVE        D1,A0
                RTS
```

Basic block 6:

```
        sFac.1: CMP         #0,D0
                BNE         sFac.2
```

Basic block 7:

Basic block 8:

```
        m.2:    MOVE        #1,D0
```

```
                        RTS
```

Basic block 9:

```
              sFac.2:   MOVE        D0,(A4)+
                        SUB         #1,D0
                        JSR         sFac.1
```

Basic block 10:

```
                        MUL         -(A4),D0
                        RTS
```

A.1.9. MC68020 code of fac.

Basic block 1:

```
lFac:                                   ; apply entry : 'Fac' node in register A2,
                                          argument n node in register A1 and node
                                          to be overwritten by result on the A-
                                          stack in register A0
      BRA       m.1                     ; jump to label m.1
```

Basic block 2:

```
nFac:                                   ; node entry : 'Fac n' node in register A0
      MOVEA.L   8(A0),A1                ; move pointer to argument part of node to
                                          A1
      MOVEA.L   (A1),A1                 ; move argument n to A1
```

Basic block 3:

```
m.1:  LEA       _cycle_in_spine,A2      ; load address of _cycle_in_spine code in
                                          A2
      MOVE.L    A2,(A0)                 ; store  _cycle_in_spine  as  evaluation
                                          address to detect cycles in the spine of
                                          the graph
      MOVE.L    (A1),D6                 ; load reduction address of n
      BEQ       eval_0                  ; branch if reduction address = _hnf (0)
      MOVE.L    A0,-(A3)                ; save register A0
      MOVEA.L   A1,A0                   ; move argument n node to A0
      MOVEA.L   D6,A1                   ; move evaluation address of n to address
                                          register
      JSR       (A1)                    ; evaluate argument n node
      MOVEA.L   A0,A1                   ; move  address  of  evaluated  argument  n
                                          node to A1
      MOVEA.L   (A3)+,A0                ; load register A0
```

Basic block 4:

```
eval_0:
      MOVE.L    A0,-(A3)                ; push node to be overwritten by result on
                                          the A-stack
      MOVE.L    8(A1),D0                ; load evaluated integer n in D0
      BSR       sFac.1                  ; call strict entry of Fac to compute Fac
                                          n in D0
```

Basic block 5:

```
      MOVEA.L   (A3)+,A0                ; load address of node to be overwritten
                                          by result in A0
      MOVE.L    A0,D1                   ; copy address of result node to D1
      CLR.L     (A0)+                   ; overwrite  node  to  store  result,  first
                                          store evaluation address,
```

```
      MOVE.L    #!INT+0,(A0)+          ; then the descriptor of an integer node,
      MOVE.L    D0,(A0)                ; and finally the integer in D0 (the
                                         result of Fac n)
      MOVEA.L   D1,A0                  ; return the address of the result node in
                                         A0
      RTS                              ; return, with result in A0
```

Basic block 6:

```
sFac.1:
      TST.L     D0                     ; n equal 0 ?
      BNE       sFac.2                 ; no, jump to label sFac.s
```

Basic block 7:

Basic block 8:

```
m.2:
      MOVEQ     #1,D0                  ; return 1 in D0
      RTS                              ; return
```

Basic block 9:

```
sFac.2:
      MOVE.L    D0,(A4)+               ; push n on the B-stack
      SUBQ.L    #1,D0                  ; subtract one from n in D0
      BSR       sFac.1                 ; call strict entry of Fac to compute Fac
                                         (--I n) in D0
```

Basic block 10:

```
      MULS.L    -(A4),D0               ; multiply result by n to compute Fac n
      RTS                              ; return, with result in D0
```

A.2. Append.

A.2.1. Append in Clean.

```
::    Append ![x] [x]          -> [x] ;
      Append [h | t] list      -> [h | Append t list]  |
      Append [] list           -> list ;
```

A.2.2. ABC code of append.

```
      .o 3 0
lAppend:                               || apply entry : 'Append l1' node, (l1 is
                                          the first argument) argument list node
                                          and node to be overwritten by result on
                                          the A-stack
      repl_args      1 1               || replace 'Append l1' node by l1 node on
                                          the A-stack
      jmp            m.7               || jump to label m.7
      .o 1 0
nAppend:                               || node entry: 'Append l1 list' node on
                                          the A-stack.
      push_args      0 2 2             || push arguments l1 and list of the
                                          'Append l1 list' node on the A-stack
```

```
m.7:
      set_entry      _cycle_in_spine 2     || overwrite evaluation address of the
                                             node to be overwritten by result to
                                             detects cycles in the spine of the
                                             graph
      jsr_eval                             || evaluate l1 node
      .o 3 0
sAppend.1:                                 || strict entry : evaluated l1 node, list
                                             node and node to be overwritten by
                                             result on the A-stack.
      eq_desc        _Cons 2 0             || test if l1 node is a Cons node
      jmp_true       m.8                   || yet, jump to label m.8
      jmp            sAppend.2             || no, jump to label sAppend.2
m.8:
      push_args      0 2 2                 || push h(ead) and t(ail) of l1 node on
                                             the A-stack
      create                               || create an empty node
      push_a         4                     || fill the created empty node with
                                             'Append t list'
      push_a         3
      fill           Append 2 nAppend 2
      push_a         1                     || fill the node which has to be
                                             overwritten by the
      fill           _Cons 2 _hnf 6        || result with a 'Cons h (Append t list)'
                                             node
      pop_a          4                     || pop h, t, l1 and list from the stack
      .d 1 0
      rtn                                  || return with 'Cons h (Append t list)'
                                             node on the A-stack
sAppend.2:
      eq_desc        _Nil 0 0              || test if l1 node is a Nill node
      jmp_true       m.9                   || yes, jump to label m.9
      jmp            sAppend.3             || no, jump to label sAppend.3
m.9:
      pop_a          1                     || pop l1 node from the A-stack
      jsr_eval                             || evaluate list node
      fill_a         0 1                   || overwrite node to be overwritten by
                                             result by evaluated list node
      pop_a          1                     || pop list node
      .d 1 0
      rtn                                  || return with evaluated argument list on
                                             the A-stack
sAppend.3:
      print          "Runtime Error: Rule Append of Module nfiblist does not
                                             match\n"
      halt
```

A.2.3. Basic blocks of the ABC code of append.

Basic block 1:

```
              .o              3 0
lAppend:                      repl_args1 1
              jmp             m.7
```

Basic block 2:

```
              .o              1 0
nAppend:
              push_args       0 2 2
```

Basic block 3:

```
              m.7:    set_entry       _cycle_in_spine 2
                      jsr_eval
```

Basic block 4:

```
                .o              3 0
sAppend.1:
                eq_desc         _Cons 2 0
                jmp_true        m.8
```

Basic block 5:

```
                jmp             sAppend.2
```

Basic block 6:

```
m.8:    push_args       0 2 2
        create
        push_a          4
        push_a          3
        fill            Append 2 nAppend 2
        push_a          1
        fill            _Cons 2 _hnf 6
        pop_a           4
        .d              1 0
        rtn
```

Basic block 7:

```
sAppend.2:
        eq_desc         _Nil 0 0
        jmp_true        m.9
```

Basic block 8:

```
        jmp             sAppend.3
```

Basic block 9:

```
m.9:    pop_a           1
        jsr_eval
        fill_a          0 1
        pop_a           1
        .d              1 0
        rtn
```

Basic block 10:

```
sAppend.3:
        print           "Runtime Error: Rule Append of Module nfiblist
                           does not match\n"
```

Basic block 11:

```
        halt
```

A.2.4. Intermediate code of append after stack access optimization and jump optimization.

Basic block 1:

```
 lAppend:   MOVE        8(A2),A2
            MOVE        0(A2),A2
            JMP         m.7
```

Basic block 2:

```
 nAppend:   MOVE        8(A0),A1
            MOVE        (A1)+,D0
            MOVE        0(A1),D1
            MOVE        D1,A1
            MOVE        D0,A2
```

Basic block 3:

```
m.7:        MOVE        A1,0(A4)
            LEA         _cycle_in_spine,A1
            MOVE        A1,(A0)
            MOVE        0(A4),A1
            MOVE        0(A2),D6
            BEQ         eval_3
            MOVE        A1,-(A3)
            MOVE        A0,-(A3)
            MOVE        A2,A0
            MOVE        D6,A1
            JSR         0(A1)
            MOVE        A0,A2
            MOVE        (A3)+,A0
            MOVE        (A3)+,A1
```

Basic block 4:

```
sAppend.1:
eval_3:     CMPW        #!_Cons+2,6(A2)
            BNE         sAppend.2
```

Basic block 5:

```
m.8:        MOVE        8(A2),A2
            MOVE        (A2)+,D0
            MOVE        0(A2),D1
            MOVE        A6,D2
            MOVE        D1,(A6)+
            MOVE        A1,(A6)+
            LEA         nAppend,A1
            MOVE        A6,D1
            MOVE        A1,(A6)+
            MOVE        #!Append+2,(A6)+
            MOVE        D2,(A6)+
            MOVE        A6,D2
            MOVE        D0,(A6)+
            MOVE        D1,(A6)+
            MOVE        A0,D0
            MOVE        #0,(A0)+
            MOVE        #!_Cons+2,(A0)+
            MOVE        D2,0(A0)
            MOVE        D0,A0
            RTS
```

Basic block 6:

```
sAppend.2:  CMPW        #!_Nil+0,6(A2)
            BNE         sAppend.3
```

Basic block 7:

```
m.9:        MOVE        0(A1),D6
            BEQ         eval_4
            MOVE        A0,-(A3)
            MOVE        A1,A0
            MOVE        D6,A1
            JSR         0(A1)
            MOVE        A0,A1
            MOVE        (A3)+,A0
```

Basic block 8:

```
eval_4:     MOVE        (A1)+,D0
            MOVE        (A1)+,D1
            MOVE        0(A1),D2
            MOVE        A0,D3
            MOVE        D0,(A0)+
            MOVE        D1,(A0)+
            MOVE        D2,0(A0)
            MOVE        D3,A0
```

```
                RTS
```

Basic block 9:

```
sAppend.3:      MOVE        A0,-(A3)
                MOVE        A1,-(A3)
                MOVE        A2,-(A3)
                LEA         address_of_the_string,A0
                JSR         print
```

Basic block 10:

```
                JMP         halt
```

A.2.5. MC68020 code of append.

Basic block 1:

```
lAppend:        MOVEA.L     8(A2),A2
                MOVEA.L     (A2),A2
                BRA         m.7
```

Basic block 2:

```
nAppend:        MOVEA.L     8(A0),A1
                MOVE.L      (A1)+,D0
                MOVE.L      (A1),D1
                MOVEA.L     D1,A1
                MOVEA.L     D0,A2
```

Basic block 3:

```
m.7:            MOVE.L      A1,(A4)
                LEA         _cycle_in_spine,A1
                MOVE.L      A1,(A0)
                MOVEA.L     (A4),A1
                MOVE.L      (A2),D6
                BEQ         eval_3
                MOVE.L      A1,-(A3)
                MOVE.L      A0,-(A3)
                MOVEA.L     A2,A0
                MOVEA.L     D6,A1
                JSR         (A1)
                MOVEA.L     A0,A2
                MOVEA.L     (A3)+,A0
                MOVEA.L     (A3)+,A1
```

Basic block 4:

```
sAppend.1:
eval_3:         CMP.W       #!_Cons+2,6(A2)
                BNE         sAppend.2
```

Basic block 5:

```
m.8:            SUBQ.L      #7,D7
                BMI         c_gc_1
r_gc_1:         MOVEA.L     8(A2),A2
                MOVE.L      (A2)+,D0
                MOVE.L      (A2),D1
                MOVE.L      A6,D2
                MOVE.L      D1,(A6)+
                MOVE.L      A1,(A6)+
                LEA         nAppend,A1
                MOVE.L      A6,D1
                MOVE.L      A1,(A6)+
                MOVE.L      #!Append+2,(A6)+
                MOVE.L      D2,(A6)+
                MOVE.L      A6,D2
```

```
                    MOVE.L      D0,(A6)+
                    MOVE.L      D1,(A6)+
                    MOVE.L      A0,D0
                    CLR.L       (A0)+
                    MOVE.L      #!_Cons+2,(A0)+
                    MOVE.L      D2,(A0)
                    MOVEA.L     D0,A0
                    RTS
```

Basic block 6:

```
sAppend.2:  CMP.W       #!_Nil+0,6(A2)
            BNE         sAppend.3
```

Basic block 7:

```
m.9:        MOVE.L      (A1),D6
            BEQ         eval_4
            MOVE.L      A0,-(A3)
            MOVEA.L     A1,A0
            MOVEA.L     D6,A1
            JSR         (A1)
            MOVEA.L     A0,A1
            MOVEA.L     (A3)+,A0
```

Basic block 8:

```
eval_4:     MOVE.L      (A1)+,D0
            MOVE.L      (A1)+,D1
            MOVE.L      (A1),D2
            MOVE.L      A0,D3
            MOVE.L      D0,(A0)+
            MOVE.L      D1,(A0)+
            MOVE.L      D2,(A0)
            MOVEA.L     D3,A0
            RTS
```

Basic block 9:

```
sAppend.3:  MOVE.L      A0,-(A3)
            MOVE.L      A1,-(A3)
            MOVE.L      A2,-(A3)
            LEA         address_of_the_string,A0
            JSR         print
```

Basic block 10:

```
            JMP         halt

c_gc_1:     JSR         collect_3
            BRA         r_gc_1
```

A.3. Inc.

A.3.1. Inc in Clean.

```
::    Inc ! (!INT,!INT,!INT,!INT)   ->    (!INT,      !INT,       !INT,       !INT) ;
      Inc (a, 9, 9, 9)              ->    (++I a,     0,          0,          0) |
      Inc (a, b, 9, 9)              ->    (a,         ++I b,      0,          0) |
      Inc (a, b, c, 9)              ->    (a,         b,          ++I c,      0) |
      Inc (a, b, c, d)              ->    (a,         b,          c,          ++I   d)
;
```

A.3.2. Basic blocks of the ABC code of inc.

Basic block 1:

```
        .o              3 0
lInc:   pop_a           1
        jmp             m.34
```

Basic block 2:

```
        .o              1 0
nInc:   push_args       0 1 1
```

Basic block 3:

```
m.34:   set_entry       _cycle_in_spine 1
        jsr_eval
```

Basic block 4:

```
        push_args       0 4 4
        push_a          3
        jsr_eval
```

Basic block 5:

```
        pop_a           1
        push_a          2
        jsr_eval
```

Basic block 6

```
        pop_a           1
        push_a          1
        jsr_eval
```

Basic block 7:

```
        pop_a           1
        jsr_eval
```

Basic block 8:

```
        pushI_a         3
        pushI_a         2
        pushI_a         1
        pushI_a         0
        pop_a           5
        .d              0 4 i i i i
        jsr             sInc.1
```

Basic block 9:

```
        .o              0 4 i i i i
        create
        fillI_b         3 0
        create
        fillI_b         2 0
        create
        fillI_b         1 0
        create
        fillI_b         0 0
        fill            _Tuple 4 _hnf 4
        pop_b           4
        .d              1 0
        rtn
```

Basic block 10:

```
        .o              0 4 i i i i
sInc.1:
        eqI_b           +9 1
        jmp_true        m.35
```

Basic block 11:

```
                jmp             sInc.2
```

Basic block 12:

```
        m.35:   eqI_b           +9 2
                jmp_true        m.36
```

Basic block 13:

```
                jmp             sInc.2
```

Basic block 14:

```
        m.36:   eqI_b           +9 3
                jmp_true        m.37
```

Basic block 15:

```
                jmp             sInc.2
```

Basic block 16:

```
        m.37:   pushI           +0
                pushI           +0
                pushI           +0
                push_b          3
                incI
                update_b        3 7
                update_b        2 6
                update_b        1 5
                update_b        0 4
                pop_b           4
                .d              0 4 i i i i
                rtn
```

Basic block 17:

```
        sInc.2: eqI_b           +9 2
                jmp_true        m.38
```

Basic block 18:

```
                jmp             sInc.3
```

Basic block 19:

```
        m.38:   eqI_b           +9 3
                jmp_true        m.39
```

Basic block 20:

```
                jmp             sInc.3
```

Basic block 21:

```
        m.39:   pushI           +0
                pushI           +0
                push_b          3
                incI
                update_b        2 6
                update_b        1 5
                update_b        0 4
                pop_b           3
                .d              0 4 i i i i
                rtn
```

Basic block 22:

```
        sInc.3: eqI_b           +9 3
                jmp_true        m.40
```

Basic block 23:

```
                jmp             sInc.4
```

Basic block 24:

```
        m.40:   pushI           +0
                push_b          3
                incI
                update_b        1 5
                update_b        0 4
                pop_b           2
                .d              0 4 i i i i
                rtn
```

Basic block 25:

```
        sInc.4:
                push_b          3
                incI
                update_b        0 4
                pop_b           1
                .d              0 4 i i i i
                rtn
```

Basic block 26:

```
        sInc.5: print           "Runtime Error: Rule Inc of Module masmind
                                   does not match\n"
```

Basic block 27:

```
                halt
```

A.3.3. Intermediate code of inc after stack access optimization and jump optimization.

Basic block 1:

```
         lInc:      JMP         m.34
```

Basic block 2:

```
         nInc:      MOVE        8(A0),A1
                    MOVE        0(A1),A1
```

Basic block 3:

```
         m.34:      LEA         _cycle_in_spine,A2
                    MOVE        A2,0(A0)
                    MOVE        0(A1),D6
                    BEQ         eval_107
                    MOVE        A0,-(A3)
                    MOVE        A1,A0
                    MOVE        D6,A1
                    JSR         0(A1)
                    MOVE        A0,A1
                    MOVE        (A3)+,A0
```

Basic block 4:

```
         eval_107:  MOVE        A0,-(A3)
                    MOVE        A1,-(A3)
                    MOVE        8(A1),A0
                    MOVE        (A0)+,D0
                    MOVE        (A0)+,D1
                    MOVE        (A0)+,D2
                    MOVE        0(A0),D3
                    MOVE        D3,-(A3)
                    MOVE        D2,-(A3)
                    MOVE        D3,A2
                    MOVE        D1,A0
                    MOVE        D0,A1
```

```
                MOVE        0(A2),D6
                BEQ         eval_108
                MOVE        A1,-(A3)
                MOVE        A0,-(A3)
                MOVE        A2,A0
                MOVE        D6,A1
                JSR         0(A1)
                MOVE        A0,A2
                MOVE        (A3)+,A0
                MOVE        (A3)+,A1
```

Basic block 5:

```
eval_108:       MOVE        0(A3),A2
                MOVE        0(A2),D6
                BEQ         eval_109
                MOVE        A1,-(A3)
                MOVE        A0,-(A3)
                MOVE        A2,A0
                MOVE        D6,A1
                JSR         0(A1)
                MOVE        A0,A2
                MOVE        (A3)+,A0
                MOVE        (A3)+,A1
```

Basic block 6:

```
eval_109:       MOVE        A0,A2
                MOVE        0(A2),D6
                BEQ         eval_110
                MOVE        A1,-(A3)
                MOVE        A0,-(A3)
                MOVE        A2,A0
                MOVE        D6,A1
                JSR         0(A1)
                MOVE        A0,A2
                MOVE        (A3)+,A0
                MOVE        (A3)+,A1
```

Basic block 7:

```
eval_110:       MOVE        0(A1),D6
                BEQ         eval_111
                MOVE        A0,-(A3)
                MOVE        A1,A0
                MOVE        D6,A1
                JSR         0(A1)
                MOVE        A0,A1
                MOVE        (A3)+,A0
```

Basic block 8:

```
eval_111:       MOVE        8(A0),D2
                MOVE        8(A1),D3
                MOVE        4(A3),A0
                MOVE        8(A0),D0
                MOVE        (A3),A0
                MOVE        8(A0),D1
                ADD         #12,A3
                JSR         sInc.1
```

Basic block 9:

```
                MOVE        A6,D4
                MOVE        #0,(A6)+
                MOVE        #!INT+0,(A6)+
                MOVE        D3,(A6)+
                MOVE        A6,D3
                MOVE        #0,(A6)+
```

```
                MOVE        #!INT+0,(A6)+
                MOVE        D2,(A6)+
                MOVE        A6,D2
                MOVE        #0,(A6)+
                MOVE        #!INT+0,(A6)+
                MOVE        D1,(A6)+
                MOVE        A6,D1
                MOVE        #0,(A6)+
                MOVE        #!INT+0,(A6)+
                MOVE        D0,(A6)+
                MOVE        A6,D0
                MOVE        D4,(A6)+
                MOVE        D3,(A6)+
                MOVE        D2,(A6)+
                MOVE        D1,(A6)+
                MOVE        (A3)+,A0
                MOVE        A0,D1
                MOVE        #0,(A0)+
                MOVE        #!_Tuple+4,(A0)+
                MOVE        D0,0(A1)
                MOVE        D1,A0
                RTS
```

Basic block 10:

```
sInc.1:         MOVE        #9,D4
                CMP         D2,D4
                BNE         sInc.2
```

Basic block 11:

Basic block 12:

```
m.35:           MOVE        #9,D4
                CMP         D1,D4
                BNE         sInc.2
```

Basic block 13:

Basic block 14:

```
m.36:           MOVE        #9,D4
                CMP         D0,D4
                BNE         sInc.2
```

Basic block 15:

Basic block 16:

```
m.37:           MOVE        #0,D0
                MOVE        #0,D1
                MOVE        #0,D2
                ADD         #1,D3
                RTS
```

Basic block 17:

```
sInc.2:         MOVE        #9,D4
                CMP         D1,D4
                BNE         sInc.3
```

Basic block 18:

Basic block 19:

```
m.38:           MOVE        #9,D4
                CMP         D0,D4
                BNE         sInc.3
```

Basic block 20:

Basic block 21:

```
m.39:       MOVE        #0,D0
            MOVE        #0,D1
            ADD         #1,D2
            RTS
```

Basic block 22:

```
sInc.3:     MOVE        #9,D4
            CMP         D0,D4
            BNE         sInc.4
```

Basic block 23:

Basic block 24:

```
m.40:       MOVE        #0,D0
            ADD         #1,D1
            RTS
```

Basic block 25:

```
sInc.4:     ADD         #1,D0
            RTS
```

Basic block 26:

```
sInc.5:     LEA         L29,A0
            JSR         print
```

Basic block 27:

```
            JMP         halt
```

A.3.4. MC68020 code of inc.

Basic block 1:

```
lInc:       BRA         m.34
```

Basic block 2:

```
nInc:       MOVEA.L     8(A0),A1
            MOVEA.L     (A1),A1
```

Basic block 3:

```
m.34:       LEA         _cycle_in_spine,A2
            MOVE.L      A2,(A0)
            MOVE.L      (A1),D6
            BEQ         eval_107
            MOVE.L      A0,-(A3)
            MOVEA.L     A1,A0
            MOVEA.L     D6,A1
            JSR         (A1)
            MOVEA.L     A0,A1
            MOVEA.L     (A3)+,A0
```

Basic block 4:

```
eval_107:   MOVE.L      A0,-(A3)
            MOVE.L      A1,-(A3)
            MOVEA.L     8(A1),A0
            MOVE.L      (A0)+,D0
            MOVE.L      (A0)+,D1
            MOVE.L      (A0)+,D2
            MOVE.L      (A0),D3
            MOVE.L      D3,-(A3)
            MOVE.L      D2,-(A3)
```

```
                MOVEA.L     D3,A2
                MOVEA.L     D1,A0
                MOVEA.L     D0,A1
                MOVE.L      (A2),D6
                BEQ         eval_108
                MOVE.L      A1,-(A3)
                MOVE.L      A0,-(A3)
                MOVEA.L     A2,A0
                MOVEA.L     D6,A1
                JSR         (A1)
                MOVEA.L     A0,A2
                MOVEA.L     (A3)+,A0
                MOVEA.L     (A3)+,A1
```

Basic block 5:

```
eval_108:       MOVEA.L     (A3),A2
                MOVE.L      (A2),D6
                BEQ         eval_109
                MOVE.L      A1,-(A3)
                MOVE.L      A0,-(A3)
                MOVEA.L     A2,A0
                MOVEA.L     D6,A1
                JSR         (A1)
                MOVEA.L     A0,A2
                MOVEA.L     (A3)+,A0
                MOVEA.L     (A3)+,A1
```

Basic block 6:

```
eval_109:       MOVEA.L     A0,A2
                MOVE.L      (A2),D6
                BEQ         eval_110
                MOVE.L      A1,-(A3)
                MOVE.L      A0,-(A3)
                MOVEA.L     A2,A0
                MOVEA.L     D6,A1
                JSR         (A1)
                MOVEA.L     A0,A2
                MOVEA.L     (A3)+,A0
                MOVEA.L     (A3)+,A1
```

Basic block 7:

```
eval_110:       MOVE.L      (A1),D6
                BEQ         eval_111
                MOVE.L      A0,-(A3)
                MOVEA.L     A1,A0
                MOVEA.L     D6,A1
                JSR         (A1)
                MOVEA.L     A0,A1
                MOVEA.L     (A3)+,A0
```

Basic block 8:

```
eval_111:       MOVE.L      8(A0),D2
                MOVE.L      8(A1),D3
                MOVEA.L     4(A3),A0
                MOVE.L      8(A0),D0
                MOVEA.L     (A3),A0
                MOVE.L      8(A0),D1
                LEA         12(A3),A3
                BSR         sInc.1
```

Basic block 9:

```
                MOVEQ       #16,D6
                SUB.L       D6,D7
                BMI         c_gc_1
r_gc_1:         MOVE.L      A6,D4
```

```
                CLR.L       (A6)+
                MOVE.L      #!INT+0,(A6)+
                MOVE.L      D3,(A6)+
                MOVE.L      A6,D3
                CLR.L       (A6)+
                MOVE.L      #!INT+0,(A6)+
                MOVE.L      D2,(A6)+
                MOVE.L      A6,D2
                CLR.L       (A6)+
                MOVE.L      #!INT+0,(A6)+
                MOVE.L      D1,(A6)+
                MOVE.L      A6,D1
                CLR.L       (A6)+
                MOVE.L      #!INT+0,(A6)+
                MOVE.L      D0,(A6)+
                MOVE.L      A6,D0
                MOVE.L      D4,(A6)+
                MOVE.L      D3,(A6)+
                MOVE.L      D2,(A6)+
                MOVE.L      D1,(A6)+
                MOVEA.L     (A3)+,A0
                MOVE.L      A0,D1
                CLR.L       (A0)+
                MOVE.L      #!_Tuple+4,(A0)+
                MOVE.L      D0,(A0)
                MOVEA.L     D1,A0
                RTS
```

Basic block 10:

```
sInc.1:     MOVEQ       #9,D4
            CMP.L       D2,D4
            BNE         sInc.2
```

Basic block 11:

Basic block 12:

```
m.35:       MOVEQ       #9,D4
            CMP.L       D1,D4
            BNE         sInc.2
```

Basic block 13:

Basic block 14:

```
m.36:       MOVEQ       #9,D4
            CMP.L       D0,D4
            BNE         sInc.2
```

Basic block 15:

Basic block 16:

```
m.37:       MOVEQ       #0,D0
            MOVEQ       #0,D1
            MOVEQ       #0,D2
            ADDQ.L      #1,D3
            RTS
```

Basic block 17:

```
sInc.2:     MOVEQ       #9,D4
            CMP.L       D1,D4
            BNE         sInc.3
```

Basic block 18:

Basic block 19:

```
m.38:       MOVEQ       #9,D4
            CMP.L       D0,D4
            BNE         sInc.3
```

Basic block 20:

Basic block 21:

```
m.39:       MOVEQ       #0,D0
            MOVEQ       #0,D1
            ADDQ.L      #1,D2
            RTS
```

Basic block 22:

```
sInc.3:     MOVEQ       #9,D4
            CMP.L       D0,D4
            BNE         sInc.4
```

Basic block 23:

Basic block 24:

```
m.40:       MOVEQ       #0,D0
            ADDQ.L      #1,D1
            RTS
```

Basic block 25:

```
sInc.4:     ADDQ.L      #1,D0
            RTS
```

Basic block 26:

```
sInc.5:     LEA         L29,A0
            JSR         print
```

Basic block 27:

```
            JMP         halt

c_gc_1:     JSR         collect_0
            BRA         r_gc_1
```

APPENDIX B: MC68020 cache case execution times.

In the following table for the addressing modes used by the code generator the following is indicated:
- the fetch effective address (FEA) time in clock cycles.
- the calculate effective address (CEA) time in clock cycles.
- the fetch immediate word effective address (FIWEA) time in clock cycles.
- the fetch immediate long effective address (FILEA) time in clock cycles.
- the calculate immediate word effective address (CIWEA) time in clock cycle.
- the number of effective address extension words.

These times can be used to calculate the cache case execution time together with the second table below.

Addressing mode:	FEA time	CEA time	FIWEA time	FILEA time	CIWEA time	extension words
`Dn`	0	0	2	4	2	0
`An`	0	0	-	-	-	0
`(An)`	4	2	4	4	2	0
`(An)+`	4	2	6	8	4	0
`-(An)`	5	2	5	7	?	0
`(d16,An)`	5	2	5	7	4	1
`(d16,PC)`	5	2	-	-	-	1
`#<data>.B`	2	-	-	-	-	1
`#<data>.W`	2	-	-	-	-	1
`#<data>.L`	4	-	-	-	-	2

The number of clock cycles needed to execute the instruction are:
(`Rn` means `An` or `Dn`, `<mea>` means memory effective address, not a register)

`MOVEQ #<data>,Dn`	2
`ADDQ #<data>,Rn`	2
`SUBQ #<data>,Rn`	2
`EXG Ry,Rx`	2
`MOVEM <ea>,register list`	8 + CIWEA time + 4 * number of registers
`MOVEM register list,<ea>`	4 + CIWEA time + 3 * number of registers
`ADD <ea>,Dn`	2 + FEA time
`ADDA <ea>,An`	2 + FEA time
`SUB <ea>,Dn`	2 + FEA time
`SUBA <ea>,An`	2 + FEA time
`CMP <ea>,Dn`	2 + FEA time
`CMPA <ea>,An`	4 + FEA time
`AND <ea>,Dn`	2 + FEA time
`OR <ea>,Dn`	2 + FEA time
`TST <ea>`	2 + FEA time
`ADD Dn,<mea>`	4 + FEA time
`SUB Dn,<mea>`	4 + FEA time
`AND Dn,<mea>`	4 + FEA time
`OR Dn,<mea>`	4 + FEA time
`ADDQ #<data>,<mea>`	4 + FEA time
`SUBQ #<data>,<mea>`	4 + FEA time
`ADDI #<data>,<mea>`	4 + FILEA time for .L otherwise 4 + FIWEA time
`ADDI #<data>,<mea>`	4 + FILEA time for .L otherwise 4 + FIWEA time.
`SUBI #<data>,<mea>`	4 + FILEA time for .L otherwise 4 + FIWEA time.
`CMPI #<data>,<mea>`	2 + FILEA time for .L otherwise 4 + FIWEA time.
`ANDI #<data>,<mea>`	4 + FILEA time for .L otherwise 4 + FIWEA time.
`ORI #<data>,<mea>`	4 + FILEA time for .L otherwise 4 + FIWEA time.
`EORI #<data>,<mea>`	4 + FILEA time for .L otherwise 4 + FIWEA time.
`EOR Dn,Dn`	2
`CLR Dn`	2

NEG Dn	2
NOT Dn	2
EOR Dn,<mea>	4 + FEA time
CLR <mea>	4 + CEA time.
NEG <mea>	4 + FEA time
NOT <mea>	4 + FEA time
LEA <ea>,An	2 + CEA time
PEA <ea>	5 + CEA time
EXT Dn	4
EXTB Dn	4
Scc Dn	4
Scc <mea>	6 + CEA time
MULS.L	43 + FIWEA time
DIVS.L	90 + FIWEA time
LSL #n,Dy	4
LSR #n,Dy	4
LSL Dx,Dy	6
LSR Dx,Dy	6
ASL #n,Dy	8
ASL Dx,Dy	8
ASR #n,Dy	6
ASR Dx,Dy	6
JMP (An)	6
JMP (d16,An)	8
JMP (d16,PC)	? (probably 8)
JSR (An)	7
JSR (d16,An)	9
JSR (d16,PC)	? (probably 9)
RTS	10
BSR	7
BRA	6
Bcc (taken)	6
Bcc.B (not taken)	4
Bcc.W (not taken)	6
Bcc.L (not taken)	6
DBcc (cc false, count not expired)	6
DBcc (cc false, count expired)	10
DBcc (cc true)	6

The number of clock cycles needed to execute the MOVE and MOVEA instructions are:

	Destination:		
Source:	An or Dn	(An) or (An)+	-(An) or (d16,An)
An or Dn	2	4	5
(An) or (An)+	6	7	7
-(An) or (d16,An) or (d16,PC)	7	8	8
#<data>.B, #<data>.W	4	6	7
#<data>.L	6	8	9

APPENDIX C: The nodes in the graph.

g	=	graph
l	=	label
r	=	register
fr	=	floating point register
d	=	displacement (offset)
i	=	integer constant
f	=	floating point constant
a	=	arity
n	=	number

Integer and address access nodes:

GREGISTER r1	global register, for example a stack pointer.
LEA l1	address of l1.
LOAD d1 r1	d1(r1), long word stored at address d1 + contents of register r1.
LOAD_I i1	#i1.
LOAD_ID d1 g1	d1(g1).
LOAD_B_ID d1 g1	zero extended byte of d1(g1).
LOAD_DES_ID d1 g1	sign extended word of d1(g1).
LOAD_DES_I l1 a1	descriptor offset. (offset of &l1 + 4 * a1)
REGISTER r1	r1.
STORE d1 r1 g1 g2	store g1 in d1(r1), g2 points to a LOAD or FLOAD node or is nil.
STORE_R r1 g1	store g1 in r1.

Integer arithmetic nodes:

ADD g1 g2	g1 + g2
AND g1 g2	g1 AND g2
ASR g1 g2	g2 >> g1 (signed)
CMP_EQ g1 g2	g2 == g1 ? -1 : 0
CMP_GT g1 g2	g2 > g1 ? -1 : 0
CMP_LT g1 g2	g2 < g1 ? -1 : 0
CNOT g1	g1 == 0 ? -1 : 0
DIV g1 g2	g2 / g1
EOR g1 g2	g1 EOR g2
LSL g1 g2	g2 << g1
LSR g1 g2	g2 >> g1 (unsigned)
MOD g1 g2	g2 % g1
MUL g1 g2	g2 * g1
OR g1 g2	g1 OR g2
SUB g1 g2	g2 - g1

Floating point access nodes:

FLOAD d1 r1	d1(r1), floating point value stored at address d1 + contents of register r1.
FLOAD_I f1	#f1.
FLOAD_ID d1 g1	d1(g1).
FREGISTER fr1	fr1.
FSTORE d1 r1 g1 g2 g3	store g1 in d1(r1), g2 points to a LOAD or FLOAD node or is nil, g3 points to a LOAD or FLOAD node is nil.

Floating point arithmetic nodes:

FACOS g1	acos (g1).
FADD g1 g2	g2 + ,g1.
FASIN g1	asin (g1).
FATAN g1	atan (g1).
FCMP_EQ g1 g2	g2 == g1 ? -1 : 0.
FCMP_GT g1 g2	g2 > g1 ? -1 : 0.
FCMP_LT g1 g2	g2 < g1 ? -1 : 0.
FCOS g1	cos (g1).
FDIV g1 g2	g2 / g1.
FEXP g1	exp (g1).
FHIGH g1	high long word of floating point number g1.
FITOR g1	floating point value of integer g1.
FJOIN g1 g2	floating point value with high long word g1 and low long word g2.
FLN g1	ln (g1).
FLOW g1	low long word of floating point number g1.
FMUL g1 g2	g2 * g1.
FLOG10 g1.	log10 (g1).
FREM g1 g2	g2 % g1.
FRTOI g1	integer value of rounded floating point number g1.
FSIN g1	sin (g1).
FSQRT g1	sqrt (g1).
FSUB g1 g2	g2 - g1.
FTAN g1	tan (g1).

Graph manipulation nodes:

ALLOCATE gc ga g1 .. gn	create a node containing the gc long words at address ga and g1, . . , gn
CREATE g1 .. gn	create a node containing g1, .., gn.
CREATE_R g1	create a node containing floating point number g1.
FILL g g1 .. gn	overwrite node (or first part of node) g with g1, .., gn.
MOVEM d1 g g1 .. gn	n values starting at d1(g), g1 .. gn point to the MOVEMI nodes.
MOVEMI g1	one of the values represented by cdag g1 with as root a MOVEM node.

APPENDIX D: The instructions of the intermediate code.

```
ADD a1 a1
AND a1 a1
ASR a1 a1
BMOVE a1 a2 a3
CMP a1 a2
CMPW a1 a2
DIV a1 a2
EOR a1 a2
EXG r1 r2
EXT a1
LEA a1 a2
LSL a1 a2
LSR a1 a2
MOD a1 a2 a3
MOVE a1 a2
MOVEB a1 a2
MOVEM a1 r1 .. rn
MOVEM r1 .. rn a1
MOVEW a1 a2
MUL a1 a2
OR a1 a2
SUB a1 a2
TST a1

FACOS f1 f2
FADD f1 f2
FASIN f1 f2
FATAN f1 f2
FCMP f1 f2
FCOS f1 f2
FDIV f1 f2
FEXP f1 f2
FLN f1 f2
FLOG10 f1 f2
FMUL f1 f2
FREM f1 f2
FSIN f1 f2
FSQRT f1 f2
FSUB f1 f2
FTAN f1 f2
FTST f1 f2
FMOVE f1 f2
FMOVEL a1 f1
FMOVEL f1 a1

JMP a1
JSR a1
RTS

BEQ l1
BGE l1
BGT l1
BLE l1
BLT l1
BNE l1
FBEQ l1
FBGE l1
FBGT l1
FBLE l1
FBLT l1
FBNE l1

SEQ r1
SGE r1
SGT r1
SLE r1
SLT r1
SNE r1
FSEQ r1
FSGE r1
FSGT r1
FSLE r1
FSLT r1
FSNE r1
```

APPENDIX E: Object file format.

An object file generated by the code generator for the Macintosh consists of a sequence of object file records. These records are in the data fork of the object file. Records can be recognized by the first byte. If a record consists of an odd number of bytes, a 0 byte is added to maintain word alignment.

A module is a contiguous region of memory that contains code or static data. A label is a location (offset) within a module. A segment is a collection of modules. There are two sections: a code section and a data section, code is stored in the code section, data in the data section.

All labels are given a unique, positive, 16-bit IDs. An ID is file-relative and identifies a label within one object file. IDs may be local or external. External IDs have a name.

At any given point in an object file there is one current module. New modules begin at the begin of an object file and after new module records.

A string consists of a length byte and the characters in the string.

The object file records are:

BEGIN RECORD:

Byte	1
Byte	0
Word	number of IDs

The first record in an object file must be a begin record.
The number of IDs field contains the highest ID number + 1 which occurs in this object file.

END RECORD:

Byte	2

The last record in an object file must be an end record.

IMPORT RECORD:

Byte	3
Byte	0
Word	ID
String	name

An import record associates a name with an ID. The ID may not appear in another import or label record. The import record need not appear before the ID is referenced in a reference or difference reference record.

LABEL RECORD:

Byte	4
Byte	flags
Word	ID
<Word	offset>
<String	name>

bit 0 of flags	= 1	the label appears in the data section.
	= 0	the label appears in the code section.
bit 1 of flags	= 1	an external label (with a name)
	= 0	a local label (without a name)

bit 2 of flags = 1 the offset is specified in the record
= 0 the offset is the current offset

A label record defines an ID. (label) The offset is relative to the start of the module and may be outside the module. The ID may not appear in another label or import record. The label record need not appear before the ID is referenced in a reference or difference reference record.

CODE RECORD

Byte 5
Byte size
Bytes code

Code records specify the contents of code sections. The code consists of size bytes. The code bytes are stored in the code section at the current code offset in the current module. Size is added to the current code offset.

DATA RECORD

Byte 6
Byte size
Bytes data

Data records specify the contents of data sections. The data consists of size bytes. The data bytes are stored in the data section at the current offset in the current module. Size is added to the current data offset.

REFERENCE RECORD

Byte 7
Byte flags
Word ID
<Word offset>

Bit 0 of flags = 1 the reference is from the data section.
= 0 the reference is from the code section.
Bit 1 of flags = 1 the location modified is a long word.
= 0 the location modified is a word.
Bit 2 of flags = 1 the reference is A5-relative.
= 0 the reference is not A5-relative.
Bit 3 of flags = 1 the offset is specified in the record.
= 0 the offset is the current offset.

A reference record specifies a reference to an ID. (label) The reference is from the current module. The ID field specifies the ID being referenced.

References fall into four categories:

- Code-to-code references:

  If the A5-relative flag is 1, the A5-relative offset of a jump-table entry associated with the specified label is added to the specified location.

  If the A5-relative flag is 0, the linker selects either PC-relative or A5-relative addressing. The immediately preceding word is assumed to contain a JSR, JMP, LEA or PEA instruction, and is modified to indicate either PC-relative or A5-relative addressing. If the referenced label and the current module are in the same segment, the PC-relative offset of the label is added to the contents of the specified location. If they are in different segments, the A5-relative offset of a jump-table entry associated with the specified label is added to the specified location.

- Code-to-data references:

  The A5-relative flag must be 1 for code-to-data-references. The A5-relative offset of the specified label is added to the contents of the specified location.

- Data-to-code references:

  If the A5-relative flag is 1, the A5-relative offset of a jump-table entry is added to the specified location.
  If the A5-relative flag is 0, the memory address of a jump-table entry associated with the specified label is added to the contents of the specified location, which must be a long word.

- Data-to-data references:

  If the A5-relative flag is 1, the A5-relative offset of the label is added to the specified location.
  If the A5-relative flag is 0, the memory address of the specified label is added to the contents of the specified location, which must be a long word.

UNINITIALIZED DATA

Byte	8
Byte	size

Uninitialized data records specify uninitialized data. Size is added to the current data offset.

DIFFERENCE REFERENCE RECORD

Byte	9
Byte	flags
Word	ID1
Word	ID2
<Word	offset>

Bit 0 of flags = 1	the reference appears in the data section.
= 0	the reference appears in the code section.
Bit 2 of flags = 1 and bit 1 of flags = 0	the location modified is a long word.
Bit 2 of flags = 0 and bit 1 of flags = 1	the location modified is a word.
Bit 2 of flags = 0 and bit 1 of flags = 0	the location modified is a byte.
Bit 3 of flags = 1	the offset is specified in the record
= 0	the offset is the current offset

A difference reference record specifies a reference which is the difference of two IDs. (labels)
If ID1 specifies a code reference, ID2 must also be a code reference in the same module. If ID1 is a data reference, ID2 must also be a data reference.
The value of the address of ID1 minus the address of ID2 is added to the contents of the specified location. Multiple references to the same or overlapping locations are permitted.

NEW MODULE

Byte	10

A new module record defines the start of a new module. New modules are also started at the begin of a new object file. (Not yet implemented)

REFERENCES.

Aho A. V., Johnson S. C. and Ullman. J. D., (1977). 'Code generation for expressions with common subexpressions'. *J. ACM* 24 : 1, 146-160.

Aho A. V., Sethi R., Ullman J. D., (1986). 'Compilers, Principles, Techniques and Tools', Bell Telephone Laboratories, Incorporated, Addison-Wesley.

Barendregt H.P., Eekelen M.C.J.D. van, Glauert J.R.W., Kennaway J.R., Plasmeijer M.J., Sleep M.R., (1987). 'Towards an Intermediate Language based on Graph Rewriting'. Proceedings of Parallel Architectures and Languages Europe (PARLE), part II, Eindhoven, The Netherlands. *Springer Lec. Notes Comp. Sci.* 259, 159-175.

Belady L.A., (1966). A study of replacement algorithms for a virtual storage computer, *IBM Systems J.* 5:2, 78-101.

Bruno J., Sethi R., (1976). 'Code generation for a one-register machine', *J. ACM* 22 : 3, 382-396.

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 Lec. Notes on Comp.Sci.* 274, 364 - 384.

Eekelen M.C.J.D. van, Nöcker E.G.J.M.H., Plasmeijer M.J., Smetsers J.E.W., (1989). 'Concurrent Clean'. University of Nijmegen. Technical Report 89-18.

Heerink G., (1990). 'Enkele benchmarks voor functionele talen', Technical Report, University of Nijmegen. (in Dutch)

Koopman P.W.M., Eekelen M.C.J.D. van, Nöcker E.G.J.M.H., Smetsers S., Plasmeijer M.J. (1990). 'The ABC-machine: A Sequential Stack-based Abstract Machine For Graph Rewriting'. Technical Report, University of Nijmegen.

Milner R.A., (1978). Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, Vol. 17, no. 3, 348 - 375.

Motorola, (1985,1984). 'MC68020 32-Bit Microprocessor User's Manual', Second edition, Motorola, Prentice Hall.

Mycroft A., (1984). 'Polymorphic type schemes and recursive definitions'. Proc. of the 6th Int. Conf. on Programming, *Springer Lec. Notes Comp. Sci.* 167, 217 - 228.

Nöcker E.G.J.M.H. (1988). 'Strictness Analysis based on Abstract Reduction of Term Graph Rewrite Systems. in Proceedings of the Workshop on Implementation of Lazy Functional Languages', University of Göteborg and Chalmers University of Technology, Programming Methodology Group, Report 53.

Nöcker E.G.J.M.H., (1989). 'The PABC Simulator, v0.5. Implementation Manual'. University of Nijmegen, Technical Report 89-19.

Nöcker E.G.J.M.H., Smetsers J.E.W., Eekelen M.C.J.D. van, Plasmeijer M.J., (1991). 'Concurrent Clean', submitted to the conference on Parallel Architectures and Languages Europe (Parle'91).

Plasmeijer M.J., Eekelen M.C.J.D. van (1989). Functional Programming and Parallel Graph Rewriting. Lecture notes, University of Nijmegen, to appear at Addison Wesley 1991.

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

Smetsers J.E.W., Eekelen M.C.J.D. van, Plasmeijer, M.J., (1991). 'Operational semantics of Concurrent Clean'. University of Nijmegen. Technical Report: in preparation.

Weijers G.A.H., (1990). 'Efficient Implementation of the ABC-Machine on the Sun-3, version 0.5', University of Nijmegen, Technical Report.