Operational Machine Specification in a Functional Programming Language

P.W.M. Koopman+, M.C.J.D. van Eekelen and M.J. Plasmeijer
Department of computer science, University of Nijmegen,
Toernooiveld 1, 6525 ED, Nijmegen, The Netherlands

Abstract

This paper advocates the use of a functional programming language for the formal specification of (abstract) machines. The presented description method describes machines at two levels. At the bottom layer machine components and micro instructions to handle them are described by an abstract data type. The top layer describes the machine instructions in terms of these micro instructions.

The use of a functional language for this purpose has several advantages. The abstraction mechanisms offered by a functional programming language are that good that one can abstract from irrelevant details as is required for a specification language. Functional languages have a well defined semantics such that the meaning of the specification is clear as well. On the other hand they offer the advantages of a programming language: the compiler can check the specification for partial correctness eliminating e.g. type errors and unbound identifiers (errors which occur in many published descriptions). Furthermore, the specification can be executed such that one obtains a prototype implementation almost for free. Such an executable formal specification can for instance be used to investigate the dynamic behaviour of the described machine.

For a simple machine, the proposed description method is compared with several other description methods: a traditional style, a denotational semantics and a formal specification in the language Z. To show that the proposed method is indeed useful to describe large and complicated machines, the method is applied for the specification of an abstract imperative graph rewrite machine (the ABC-machine) which has been used in the construction of the compiler for the functional language Concurrent Clean.

Keywords: functional programming language, formal specification, machine specification, prototype implementation, executable specification

1 Introduction

This paper presents a method to specify (abstract) machines in a functional programming language. The use of a functional languages description formalism pairs the conciseness and abstractness of a specification language with the executability of a programming language. The use of programming languages instead of a special purpose formalism for the specification has a number of attractive advantages:

- The syntactical correctness, absence of unbound identifiers and type consistency can be checked most efficiently and concisely by a compiler. This does not imply that a compiler is able to verify that the complete specification is correct, only partial correctness can be tested.
- Using an existing programming language there is no need nor place for new language constructs in the description. This implies that the semantics of the description are clear and well defined.
- The specification is its own prototype implementation. This eliminates the effort to construct a prototype and to keep the specified product and prototype similar. Although each test shows just that the specified system behaves as shown on the test data, it yields valuable information on the dynamic properties of the system.

The obvious drawback of the use of a programming language as specification language is the great implementation effort commonly required. Much more time is spent on solving all kinds of implementation

+ Currently: Department of computer science, University of Leiden, Niels Bohrweg 1, 2333 AC, Leiden, The Netherlands details than on writing a specification of the product. The difference between the design of the algorithm and the construction of the actual product vanishes.

Functional programming languages offer a high abstraction level and powerful language constructs. So, they seem to be better suited as specification formalism than imperative programming languages. In this paper the functional language Miranda*[1] is used to present the definition of an abstract machine. An elaborated discussion of the descriptions presented here can be found in[2].

The proposed description method is first generally described. In section 3 the proposed machine specification method is compared with a traditional description of a simple machine. Then, the specification method is used for a new large scale application: the ABC-machine, an abstract imperative graph rewrite machine that has actually be used in the implementation of the functional programming language Concurrent Clean.

2 Operational machine specification

Like other semantic machine descriptions an operational machine specification consists of an description of the memory components in the machine, the *state*, and the ways to change this state: the *instructions*.

The machine **state** is the collection of all parts of the machine visible to the programmer. All these parts are memory components. This machine state is specified by an enumeration of memory components in a single construct. We have chosen to use a tuple type to construct the machine state from the states of the components.

```
state == ( mem1, … , memn )
```

Each memory component is described by an abstract data type. The access functions on this abstract data type can in this context be regarded as the **micro-instructions** of the machine. This are the primitive access functions describing the components of the machine. In the implementation of the abstract types we can put the amount of detail required, e.g. the contents of memory words can be represented as bit strings or as numbers.

```
abstype  mem1
   with  micro_instruction1.1 :: …
         micro_instruction1.2 :: …
         …
…
abstype  memn
   with  micro_instructionn.1 :: …
         …
```

Executing an instruction causes a state transition in of the machine. The **instructions** are functions transforming the current machine state in the new machine state corresponding to this instruction. Hence, the machine state is changed by applying the current instructions to the state of the machine. The type of all instructions is therefore:

```
instruction == state -> state
```

Instructions are described in a two-level model. The bottom level consists of the micro-instructions. The top level consists of the definition of the state transition of each instruction in terms of these micro-instructions. Some examples of instructions are:

```
no_op :: instruction
no_op state = state

jump :: address -> instruction
jump x ( mem1, … , pc , … memn )  =  ( mem1, … , pc' , … memn )
                                     where    pc' = pc_set x
```

Apart from the semantics of the individual instructions a complete machine specification requires also a description of which instruction is applied in a given machine state. In the programmable machines

---

* Miranda is a trade mark of Research Software Ltd. The Miranda system version 2.009 (13 November 1989) is referenced in this paper as the Miranda system.

(computers) described in this paper the order of the instructions to execute is determined by a program stored in the memory components of the machine. The machine runs by executing an instruction cycle until some final state of the machine is reached. The instruction cycle consists of extracting the current instruction from the actual machine state and applying it to the state. This clearly reflects the von Neumann machine instruction cycle.

```
instruction_cycle :: state -> state
instruction_cycle state
 =  state                                    , if is_final_state state
 =  instruction_cycle (instruction state)    , otherwise
    where   instruction  =  select_current_instruction state
```

As illustrated below the instruction cycle can be tailor made for a given machine when the type of the state is known. It is possible and useful to lift actions common to all instructions like incrementing a program counter from the specification of individual instructions to the instruction cycle. These concepts are further developed and illustrated in the rest of this paper.

3 Description of a simple machine

To compare the functional description with a conventional description we present two descriptions of the same machine. First, a conventional description of this simple machine is given. Then, the functional specification will be shown.

The machine presented here is a simple conventional machine, called Mac-1. It is used by Tanenbaum[3] to illustrate the concept of micro-programming.

3.1 Overview of the Mac-1 architecture

The Mac-1 is a small machine with a memory of 4096 16-bit words. This memory contains the program to be executed and a stack. The stack grows from high memory addresses to lower ones. The top of the stack is indicated by the stack pointer (sp). The current instruction is indicated by the program counter (pc). The machine has one register, the accumulator (ac), to store the result of computations. The architecture of Mac-1 is depicted in figure 1.

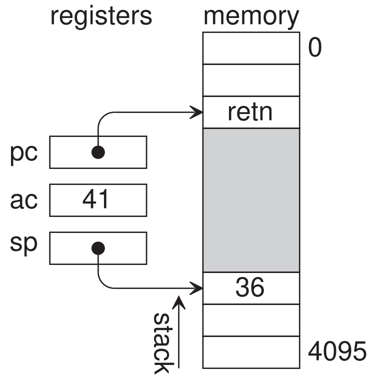


Figure 1. The architecture of Mac-1

Four addressing modes are provided in this machine:
immediate: the operand is specified in the low order bits (8 or 12) of the instruction;
direct: the low-order 12-bits of the instruction are the address of the operand;
indirect: the address of the operand is in the accumulator;
local: the operand is on the stack, the offset is given in the low-order 12-bits of the instruction.
The addressing mode is indicated by the instruction used. There is a very limited set of instructions available:
Load: load the accumulator with the specified operand;
Store: store the contents of the accumulator at the specified address in the memory;
Add: add the operand to the contents of the accumulator, the sum is stored in the accumulator;

Sub: subtract the operand from the contents of the accumulator, the result is stored in the accumulator;
Jump: set the program counter to the specified address, some jumps are conditional on the contents of the accumulator;
Push: push the specified operand on the stack;
Pop: pop an item from the stack;
Swap: exchange the contents of the accumulator and stack pointer;
Sp handling: *increment* or *decrement* the stack pointer by the 8-bit constant given in the instruction.

This machine uses memory mapped io, the machine can communicate with the world by reading and writing to specific memory locations. The output values at these locations are visible from outside, the inputs are supplied from the external world to these locations. This aspect of the machine is omitted for reasons of brevity.

3.2 The conventional description of the Mac-1 instructions

The specification of the instruction set in table 1 is taken from[3]. The meaning of the instructions is given by a Pascal-like program fragment. In these pseudo code fragments, m [x] refers to memory word x.

Binary	Mnemonic	Instruction	Meaning
0000xxxxxxxxxxxx	LODD	Load direct	ac := m [x]
0001xxxxxxxxxxxx	STOD	Store direct	m [x] := ac
0010xxxxxxxxxxxx	ADDD	Add direct	ac := ac + m [x]
0011xxxxxxxxxxxx	SUBD	Subtract direct	ac := ac - m [x]
0100xxxxxxxxxxxx	JPOS	Jump positive	**if** ac ≥ 0 **then** pc := x
0101xxxxxxxxxxxx	JZER	Jump zero	**if** ac = 0 **then** pc := x
0110xxxxxxxxxxxx	JUMP	Jump	pc := x
0111xxxxxxxxxxxx	LOCO	Load constant	ac := x (0 ≤ x ≤ 4095)
1000xxxxxxxxxxxx	LODL	Load local	ac := m [sp + x]
1001xxxxxxxxxxxx	STOL	Store local	m [x + sp] := ac
1010xxxxxxxxxxxx	ADDL	Add local	ac := ac + m [sp + x]
1011xxxxxxxxxxxx	SUBL	Subtract local	ac := ac - m [sp + x]
1100xxxxxxxxxxxx	JNEG	Jump negative	**if** ac < 0 **then** pc := x
1101xxxxxxxxxxxx	JNZE	Jump non zero	**if** ac ≠ 0 **then** pc := x
1110xxxxxxxxxxxx	CALL	Call procedure	sp := sp - 1 ; m [sp] := pc ; pc := x
1111000000000000	PSHI	Push indirect	sp := sp - 1 ; m [sp] := m [ac]
1111001000000000	POPI	Pop indirect	m [ac] := m [sp] ; sp := sp + 1
1111010000000000	PUSH	Push onto stack	sp := sp - 1; m [sp] := ac
1111011000000000	POP	Pop from stack	ac := m [sp] ; sp := sp + 1
1111100000000000	RETN	Return	pc := m [sp] ; sp := sp + 1
1111101000000000	SWAP	Swap ac, sp	tmp := ac ; ac := sp ; sp := tmp
11111100yyyyyyyy	INSP	Increment sp	sp := sp + y (0 ≤ y ≤ 255)
11111110yyyyyyyy	DESP	Decrement sp	sp := sp - y (0 ≤ y ≤ 255)

xxxxxxxxxxxx is a 12-bit machine address; in column 4 it is called x.
yyyyyyyy is an 8-bit constant; in column 4 it is called y.

Table 1. The Mac-1 instruction set as described by Tanenbaum.

3.3 The operational specification of the Mac-1

To specify the semantics of instructions it is sufficient to relate their names to their meaning, i.e. the effect on the machine state or, in other words, the relation between column two and four in table 1. A complete specification of the machine requires also a description of which instruction is selected for execution. In the Mac-1 machine this is the instruction that is encoded by the memory word addressed by the program counter. The relation between word contents and instruction is given by the first and second column of table 1. The operational specification of Mac-1 machine described below is conform the structure outlined in section 2.

The state of Mac-1

The architecture of Mac-1 contains three registers (pc, ac and sp) and a memory. The machine state is fully determined by the contents of these three registers and the memory. In the specification the state is given by the tuple (pc, ac, sp, me).

```
state  ==   (pc, ac, sp, me)
pc     ==   word
ac     ==   word
sp     ==   word
me     ==   mem
```

The micro-instruction set of Mac-1

The micro-instruction level consists of the descriptions of all individual machine components. Each component is described by the access functions defined on it; the ways it can be used.

The memory mem is defined by a small set of micro-instructions. The access functions defined are:

get a: returns the number stored at address a in the memory;

update a n: replaces the contents of the memory location at address a with the number n, yields the updated memory;

store prog: yields a new memory containing the program prog. This micro-instruction is only needed when the specification is used as prototype implementation, the instructions do not use it.

The memory is formally defined by the abstract type mem.

```
abstype mem
   with get      ::  address -> mem -> word
        update   ::  address -> word -> mem -> mem
        store    ::  program -> mem
```

The registers and the memory of Mac-1 contain 16-bit words. These words are treated as two's complement integers in arithmetic. In other situations all bit strings are treated as positive numbers. The low order 12-bits of a word can be used as an address, and a positive 12-bit word is silently converted to a 16-bit word by adding 4 zero's in the front. In order to allow a fair comparison with the description in table 1 all these bit-strings are represented by a single type word. It is very simple to use different types for the various bit-strings and add the needed conversions explicitly.

```
address ==   word

abstype word
   with add                 ::  word -> word -> word
        sub                 ::  word -> word -> word
        is_neg              ::  word -> bool
        eq                  ::  word -> word -> bool
        wordTOinstruction   ::  word -> instruction
        instrTOword         ::  instr -> word
        addressTOnum        ::  address -> num
        wordTOnum           ::  word -> num
        numTOword           ::  num -> num -> word
```

The micro-instructions are of the same level of abstraction as the Pascal fragments, like m[x], used in table 1. The informal description and type definitions are usually sufficient to understand them. Though the definitions will be seldom referenced a implementation of the type mem is included in appendix 1 for the sake of completeness and to show that such an implementation is indeed very simple. The implementation of the micro-instructions is essential for execution of entire machine specification.

The instruction set of Mac-1

The instructions change the state of the machine. This state change is described in terms of micro-instructions. The type of the describing functions are all very similar. Some examples are:

```
lodd  ::  word -> instruction
jump  ::  word -> instruction
retn  ::  instruction
```

These type definitions reflect the fact that jump on its one is not an instruction, but the combination of jump and a word (the target address) is an instruction.

The specification of the semantics of the instructions in the proposed format is given bellow. (The types of the functions are left out.)

```
lodd x  (pc, ac, sp, me) = (pc, ac', sp, me)    where  ac'  = get x me
stod x  (pc, ac, sp, me) = (pc, ac, sp, me')    where  me'  = update x ac me
addd x  (pc, ac, sp, me) = (pc, ac', sp, me)    where  ac'  = add ac (get x me)
subd x  (pc, ac, sp, me) = (pc, ac', sp, me)    where  ac'  = sub ac (get x me)
jpos x  (pc, ac, sp, me) = (pc', ac, sp, me)    where  pc'  = cond (~is_neg ac) x pc
jzer x  (pc, ac, sp, me) = (pc', ac, sp, me)    where  pc'  = cond (eq zero ac) x pc
jump x  (pc, ac, sp, me) = (pc', ac, sp, me)    where  pc'  = x
loco x  (pc, ac, sp, me) = (pc, ac', sp, me)    where  ac'  = x
lodl x  (pc, ac, sp, me) = (pc, ac', sp, me)    where  ac'  = get (add sp x) me
stol x  (pc, ac, sp, me) = (pc, ac, sp, me')    where  me'  = update (add sp x) ac me
addl x  (pc, ac, sp, me) = (pc, ac', sp, me)    where  ac'  = add ac (get (add sp x) me)
subl x  (pc, ac, sp, me) = (pc, ac', sp, me)    where  ac'  = sub ac (get (add sp x) me)
jneg x  (pc, ac, sp, me) = (pc', ac, sp, me)    where  pc'  = cond (is_neg ac) x pc
jnze x  (pc, ac, sp, me) = (pc', ac, sp, me)    where  pc'  = cond (~eq zero ac) x pc
call x  (pc, ac, sp, me) = (pc', ac, sp', me')  where  pc'  = x
                                                       sp'  = sub sp one
                                                       me'  = update sp' pc me
pshi    (pc, ac, sp, me) = (pc, ac, sp', me')   where  sp'  = sub sp one
                                                       me'  = update sp' (get ac me) me
popi    (pc, ac, sp, me) = (pc, ac, sp', me')   where  sp'  = add sp one
                                                       me'  = update ac (get sp me) me
push    (pc, ac, sp, me) = (pc, ac, sp', me')   where  sp'  = sub sp one
                                                       me'  = update sp' ac me
pop     (pc, ac, sp, me) = (pc, ac', sp', me)   where  sp'  = add sp one
                                                       ac'  = get sp me
retn    (pc, ac, sp, me) = (pc', ac, sp', me)   where  pc'  = get sp me
                                                       sp'  = add sp one
swap    (pc, ac, sp, me) = (pc, ac', sp', me)   where  ac'  = sp
                                                       sp'  = ac
insp y  (pc, ac, sp, me) = (pc, ac, sp', me)    where  sp'  = add sp y
desp y  (pc, ac, sp, me) = (pc, ac, sp', me)    where  sp'  = sub sp y
```

Table 2. The operational specification of the Mac-1 instruction set in Miranda.

Execution of Mac-1 programs

The program, the instruction sequence to be executed, is stored in the memory. The current instruction is addressed by a program counter. The current instruction is be executed by extracting the word addressed by the program counter from the memory, translating it to an instruction and applying it to the current state. The program counter is incremented before the instruction is applied to the state. By applying the instruction cycle recursively to the state delivered by the current instruction the machine continuously executes instructions. This clearly reflects the von Neumann machine instruction cycle. The instruction cycle is interrupted when the word addressed by the program counter is not an instruction.

```
instruction_cycle :: state -> state
instruction_cycle (pc, ac, sp, mem)
 =  instruction_cycle (instr (pc', ac, sp, mem))     , if is_instr
 =  (pc, ac, sp, mem)                                , otherwise
    where  pc'                =  add pc one
           word               =  get pc mem
           (is_instr, instr)  =  wordTOinstruction word
```

The function to convert a word into an instruction is part of the abstract type word. Since not all words are representations of instructions, the conversion function delivers a tuple containing a Boolean indicating whether the word is an instruction and the instruction itself.

The initial state of Mac-1

The instructions define state transitions of the machine. The instruction cycle defines the sequence of state changes. What remains to complete the description is the specification of the initial state. In order to execute programs with the specification we have added a user friendly way to store a program in the initial machine state. The program execution is chosen to start at address 0.

```
boot :: program -> state
boot program
 =  instruction_cycle (pc, ac, sp, mem)
    where  pc   =  zero
           ac   =  zero
           sp   =  initial_sp
           mem  =  store program

initial_sp  =  numTOword 4090     ||  The words at adresses 4091..4095 are used for memory mapped IO.
```

In principle it is possible to encode the instructions of the program to be executed as numbers and store these numbers as words in the memory. However, this is a tedious way to write programs. A simple data structure to represent the instructions and the corresponding conversion to words reveals this. There are constructors added to denote an non-instruction word (STOP) and to enter an arbitrary 16-bit signed number. It is straight forward to extend this to a full fledged assembler by adding labels and macro's.

```
instr ::=  LODD x  | STOD x   | ADDD x  | SUBD x  | JPOS x  | JZER x  | JUMP x  | LOCO x  |
           LODL x  | STOL x   | ADDL x  | SUBL x  | JNEG x  | JNZE x  | CALL x  | PSHI    |
           POPI    | PUSH     | POP     | RETN    | SWAP    | INSP y  | DESP y  |
           STOP    | CONS num

x == num ; y == num

program  ==  [instr]
```

Example Mac-1 program

The example Mac-1 program shown here computes Fibonacci numbers is a naive recursive way. The Fibonacci function is defined as:

```
fib :: num -> num
fib n   = 1                        , if n <= 1
        = fib (n-1) + fib ( n-2)   , otherwise
```

A program for Mac-1 to compute this function is. The argument and the result of the function are passed in the accumulator. The argument of the Fibonaci function in Mac-1 code is an argument of the Miranda function containing this code.

```
fib_program :: num -> program
fib_program n  =  [   LOCO    n       ,   ||  0      load argument in accumulator
```

```
CALL   3    ,  ||  1    call the Fibonacci function
STOP        ,  ||  2

SUBD   17   ,  ||  3    compute n-2
JNEG   14   ,  ||  4    goto 14  if n ≤ 1
PUSH        ,  ||  5    push n-2
ADDD   16   ,  ||  6    compute n-1
CALL   3    ,  ||  7    compute fib (n-1)
PUSH        ,  ||  8    push fib (n-1)
LODL   1    ,  ||  9    load n-2 in accumulator
CALL   3    ,  ||  10   compute fib (n-2)
ADDL   0    ,  ||  11   add fib (n-2) to fib (n-1)
INSP   2    ,  ||  12   clear stack
RETN        ,  ||  13   return to caller

LOCO   1    ,  ||  14   load accumulator with result
RETN        ,  ||  15   return to caller

CONS   1    ,  ||  16   the constant 1
CONS   2    ]  ||  17   the constant 2
```

Tracing the execution of Mac-1 programs

The value of the specification as a prototype implementation can be further increased by adding a way to show the dynamic behaviour of the machine. The function instruction_cycle executes Mac-1 programs, but only the final state can be observed. One of the advantages of this way of specifying machines is that it is very easy to adapt it in order to obtain a trace of the execution. The instruction_cycle is replaced by trace_cycle that has the same effect on the machine state, but yields trace information. We have chosen here to show the contents of the registers and the stack.

```
trace_cycle :: state -> [char]
trace_cycle (pc, ac, sp, me)
 =  show_state (pc, ac, sp, me) ++ rest
    where  pc'                =  add pc one
           word               =  get pc me
           (is_instr, instr)  =  wordTOinstruction word
           rest               =  trace_cycle (instr (pc', ac, sp, me))  , if is_instr
                              =  [ ]                                    , otherwise

show_state :: state -> [char]
show_state (pc, ac, sp, me)
 =  "ac=" ++ show (wordTOnum ac) ++ "\tsp=" ++ show (wordTOnum sp) ++
    "\tpc=" ++ show (wordTOnum pc) ++ "\tstack=" ++
    show [wordTOnum (get (numTOword i) me) | i <- [addressTOnum sp .. addressTOnum initial_sp-1]] ++ "\n"
```

trace of the example program

Using the function trace_cycle the execution of the Fibonacci function can be traced. The trace of the execution of the program fib_program 2 is shown below. This illustrates clearly how the Fibonacci number is computed by this program on the Mac1-machine.

```
ac = 0    sp = 4090   pc = 0    stack = [ ]
ac = 2    sp = 4090   pc = 1    stack = [ ]
ac = 2    sp = 4089   pc = 3    stack = [ 2 ]
ac = 0    sp = 4089   pc = 4    stack = [ 2 ]
ac = 0    sp = 4089   pc = 5    stack = [ 2 ]
ac = 0    sp = 4088   pc = 6    stack = [ 0, 2 ]
ac = 1    sp = 4088   pc = 7    stack = [ 0, 2 ]
ac = 1    sp = 4087   pc = 3    stack = [ 8, 0, 2 ]
ac = -1   sp = 4087   pc = 4    stack = [ 8, 0, 2 ]
ac = -1   sp = 4087   pc = 14   stack = [ 8, 0, 2 ]
ac = 1    sp = 4087   pc = 15   stack = [ 8, 0, 2 ]
```

```
ac = 1    sp = 4088   pc = 8    stack = [ 0, 2 ]
ac = 1    sp = 4087   pc = 9    stack = [ 1, 0, 2 ]
ac = 0    sp = 4087   pc = 10   stack = [ 1, 0, 2 ]
ac = 0    sp = 4086   pc = 3    stack = [ 11, 1, 0, 2 ]
ac = -2   sp = 4086   pc = 4    stack = [ 11, 1, 0, 2 ]
ac = -2   sp = 4086   pc = 14   stack = [ 11, 1, 0, 2 ]
ac = 1    sp = 4086   pc = 15   stack = [ 11, 1, 0, 2 ]
ac = 1    sp = 4087   pc = 11   stack = [ 1, 0, 2 ]
ac = 2    sp = 4087   pc = 12   stack = [ 1, 0, 2 ]
ac = 2    sp = 4089   pc = 13   stack = [ 2 ]
ac = 2    sp = 4090   pc = 2    stack = [ ]
```

Unfortunately, it is not directly possible to show the current instruction in a trace. The instructions stored in the memory of Mac-1 are partially parametrized functions. The Miranda system shows all partially parametrized functions as `<function>` and the comparison of such curried functions causes an erroneous program termination with the message `program error: attempt to compare functions`. This behaviour is suggested by the computational model used; the λ-calculus. In the λ-calculus a comparison of two functions implies a test for extensional equality: do these functions behave identical for all possible arguments. This is in general an undecidable property. A syntactical comparison can be added to the λ-calculus, it is known as Church's δ rule[4]. For the trace we are interested in the function name and not in the corresponding λ-term. So, Church's δ rule will not solve our problem.

In a rewrite system partially parametrized functions are ordinary terms that can be shown for tracing purposes. It would be better if the semantics of functional programming languages with pattern matching was based on rewrite systems. Clean is such a functional programming language[5,6], see also below.

By writing an additional conversion function from numbers to some printable object (e.g. the type instr) it is possible to show the instruction to be executed in the trace of a program. In this machine it is necessary to have conversions between numbers and instructions since both objects are stored in the same memory.

Alternative styles of specification

It is possible to give equivalent definitions for the instructions without using local definitions. This is achieved by substitution of the local defined names by the body of the definition. For instance:

```
call x (pc, ac, sp, me)  =  (x, ac, sub sp one, update (sub sp one) pc me)
```

Although the definition is shorter without local definitions, we prefer the version given in table 2 since it is more elegant and better readable. Especially for machines with an state consisting of many components the local definitions serve as useful indications of the components changed. Note that we do not include a definition of the form ac' = ac in the definition of the call instruction to indicate that the accumulator is not changed.

The fact that the state is a tuple consisting of four components and the order of these components is explicitly used in the specification. This has two reasons. First of all we want to show the state changes by the instructions very clearly. Secondly, a specification without using the structure of the state explicitly results in ugly and clumsy definitions for more complex instructions. To hide the structure of the state it must represented by an abstract data type:

```
abstype state
   with   get_pc       ::  state -> pc
          get_sp       ::  state -> sp
          get_ac       ::  state -> ac
          get_me       ::  state -> me
          update_pc    ::  pc -> state -> state
          update_sp    ::  sp -> state -> state
          update_ac    ::  ac -> state -> state
          update_me    ::  me -> state -> state
```

The definition of simple instructions becomes a little bit better using this style because components that are not involved are not mentioned in the definition. Moreover, it is much easier to add or remove a component to the state when it is represented by an abstract type. The definition of instructions that does not use this components are not changed. However, the definition of more complex instructions becomes considerably more difficult to read. This is illustrated by the definition of the instructions jump and call.

```
jump  x state   =  update_pc x state
call  x state   =  update_pc x (update_sp sp' (update_me me' state))
                   where  sp'  =  sub sp one;
                          me'  =  update sp' pc me;
                          sp   =  get_sp state;
                          pc   =  get_pc state
```

Moreover, the definition of the call instruction in this style suggest that memory (me) must be updated before the stack pointer (sp) is updated and the stack pointer must be updated before the program counter (pc) is updated. This is a very undesirable aspect since it is not true.

Instead of making each instruction a function that performs the desired state transition, it is possible to have one interpreting function describing the effect of all instructions. Using the type instr to indicate the instruction to be executed such an interpreting function can be defined as

```
execute :: instr -> state -> state
execute (LODD x) (pc, ac, sp, me)  =  (pc, ac', sp, me)    where  ac'  =  get x me
execute (STOD x) (pc, ac, sp, me)  =  (pc, ac, sp, me')    where  me'  =  update x ac me
execute (ADDD x) (pc, ac, sp, me)  =  (pc, ac', sp, me)    where  ac'  =  add ac (get x me)
execute (SUBD x) (pc, ac, sp, me)  =  (pc, ac', sp, me)    where  ac'  =  sub ac (get x me)
execute (CALL  x) (pc, ac, sp, me) =  (pc', ac, sp', me')  where  pc'  =  x
                                                                  sp'  =  sub sp one
                                                                  me'  =  update sp' pc me
…
```

Instead of one function for each instruction there is now only one interpretative function (execute) that describes the effect of all instructions. The function lodd x can be seen as the result of the partial evaluation of execute (LODD x). These forms are clearly equivalent. We prefer the style with one function per instruction since it is more modular. Instructions can be introduced and explained one by one instead of in one giant function execute.

Instead of implementing the memory as a list of words as shown in appendix 1, it is also possible to use a function that associates words to addresses. The functions get and update from the abstract type memory becomes:

```
mem  ==     address -> word

get address mem = mem address
update address new mem x = cond (eq x address) new (mem x)
```

When the execute function is combined with this type of memory the operational machine description becomes a direct executable form of denotational semantics[**refje toevoegen Gordon, Schmidt**] for this very simple language that consists of lists of instructions. All memory components in such a description are usually treated by the a single function, this can be achieved by adding an appropriate data type to address the memory components in the machine. For the instructions requiring multiple updates of the machine state an order of updates must be arbitrarily chosen.

```
state      ==    mem
mem        ==    location -> word
location   ::=   PS | AC | SP | Word word

update   ::  location -> word -> mem -> mem
update location new mem loc = cond (eq loc location) new (mem loc)

execute :: instr -> state -> state
execute (LODD x) state  =  update AC x state
execute (STOD x) state  =  update (Word x) ac state
execute (ADDD x) state  =  update AC (add (state AC) (state (Word x))) state
execute (SUBD x) state  =  update AC (sub (state AC) (state (Word x))) state
execute (CALL x) state  =  update PC x
                           (update (Word (sub (state SP) one)) (state PC)
                           (update SP (sub (state SP) one) state))
```

...

In denotational semantics the type instr is usually replaced by a match on syntax indicated by the Scott brackets Œ and ”. The argument state is curried whenever possible, when it is necessary to have it explicitly it is introduced by an lambda abstraction instead of an argument on the left hand side.

```
execute :: instr -> state -> state
execute Œ LODD  x ”    =  update AC x
execute Œ STOD  x ”    =  update (Word x) ac
execute Œ ADDD  x ”    =  λ state .  update AC (add (state AC) (state (Word x))) state
execute Œ SUBD      x ”    =  λ state .  update AC (sub (state AC) (state (Word x))) state
execute Œ CALL  x ”    =  λ state .  update PC x
                                      (( λ state' . update (Word (state' SP)) (state' PC))
                                      (update SP (sub (state SP) one) state))
...
```

Comparison with other formal specification methods

A specification of the Mac-1 machine in the specification language Z[**refje naar Spivey**] resembles the specifications given here, although the semantics of Z is different from functional programming languages. The state of the machine is given by the schema *State*. A suitable type to represented the bit strings of length 16 are the numbers between 0 and 2^16-1, it is called *Word*. The memory will be modelled by a function from numbers to *Word*.

Word == 0 .. 65535
Memory == *num* |,→ *Word*

State

pc, *ac*, *sp*: *Word*
me: *Memory*

dom *me* = 0 .. 4095

The state transitions are usually also described by schema's. The state transitions caused by two instructions are given as examples. Selecting the current instruction and updating the program counter is done outside the given schema's.

Lodd

Δ*State*
x?: *Word*

pc' = *pc*
ac' = *x*?
sp' = *sp*
me' = *me*

Call

Δ*State*
x?: *Word*

pc' = *x*?
ac' = *ac*
sp' = (*sp* - 1) mod 65536
me' = *me* ⊕ {(*sp* -1) mod 4096 Î pc }

Although it is possible to hide all manipulations of the type *Word* occurring in these schema's in functions, there is no way to prevent that instances of type *Word* are used as ordinary numbers. In the Miranda specification in table 2 this is done by using the abstract data type word. In the specification in table 1 words are treated as integer and the fact that it are actually bit strings of a finite length with some different interpretations is silently ignored.

It is also possible to see the instruction cycle as the basic state transition. This state transition can also take care of incrementing the program counter before the selected instruction is executed. In this situation it is more convenient to define the instructions as function instead of schema's. As outlined above we prefer on function for each instruction. In this situation it is necessary to group all machine components in one data type. The functions corresponding to the instructions specified in the schema's above are:

$$state \;==\; Word \times Word \times Word \times Memory$$

$$\begin{array}{|l} lodd: Word \rightarrow \mathrm{state} \rightarrow \mathrm{state} \\ \hline \forall\, x, pc, ac, sp: Word; me: Memory \bullet \\ \quad lodd\ x\ (pc, ac, sp, me) \;=\; (pc, x, sp, me) \end{array}$$

$$\begin{array}{|l} call: Word \rightarrow \mathrm{state} \rightarrow \mathrm{state} \\ \hline \forall\, x, pc, ac, sp: Word; me: Memory \bullet \\ \quad call\ x\ (pc, ac, sp, me) \;=\; (x, ac, (sp\ \text{-}1) \bmod 65536, me \oplus \{(sp\ \text{-}1) \bmod 4096\ \hat{\mathrm{I}}\ \mathrm{pc}\ \}) \end{array}$$

reflection

In this section a simple concrete machine architecture was described in several ways. The specifications in Miranda and Z can be checked automatically on type consistency. When a specification passes such a test confidence in the correctness increases. The specification in Miranda is a useful prototype implementation of the machine. When the prototype behaves as expected conviction in the accuracy of the specification is great.

The specification in table 1 is the smallest, but it is inaccurate and incomplete. It does not specify an instruction cycle and the moment of incrementing the program counter. In all other specifications the program counter is incremented after fetching the instruction and before executing it. In principle it is also possible to increment the program counter after execution of the current instruction. Also the arithmetic used in table 1 does not reflect the fact that numbers are stored in finite bit strings. This specification is the only one that needs an additional variable (tmp) in the swap instruction and needs to make an arbitrary choice to put the contents of one of the registers involved in that variable.

Compared with denotational semantics the proposed specification method takes the machine as starting point, while denotational semantics assigns a meaning to the programming language consisting of sequences of instructions. These descriptions can be transformed in each other by a number of small steps as shown above.

The given machine specification has an immediate equivalent in the language Z. Due to that lack of abstract data types in the language Z it is not possible to hide the representation of memory words while forcing and to force the proper manipulation at the same time. The need to list the type of each variable explicitly the specification in Z is somewhat longer. The specification in Z can be type checked like the Miranda specification.

The operational specification in a functional programming language is short, complete and very clear compared with the alternatives discussed. Moreover, it is the only one that can be executed to observe the dynamic behaviour of the described machine.

4 Large scale application of the description method

The description method introduced in the previous part of this paper in used successfully in a large implementation project. This part of the paper gives a global description of this project and highlights the role of used machine description within this project.

The aim of the project was to develop a highly efficient implementation of functional programming languages largely independent of the concrete language and target machine. To achieve this goal two intermediate levels are defined.

The first intermediate level is intermediate functional programming language containing the essential properties of functional programming languages, but not the syntactical sugar provided by various high level programming languages.

The second intermediate level is an abstract graph rewriting machine as an abstraction of various concrete machines. This abstract machine describes graph rewriting on a level of abstraction close to that of concrete conventional machines without being cluttered by the details of a specific hardware implementation. This abstract machine is called the ABC-machine. The letters A, B and C represent the three stacks used in this machine.

A detailed description of this translation path can be found in[7].

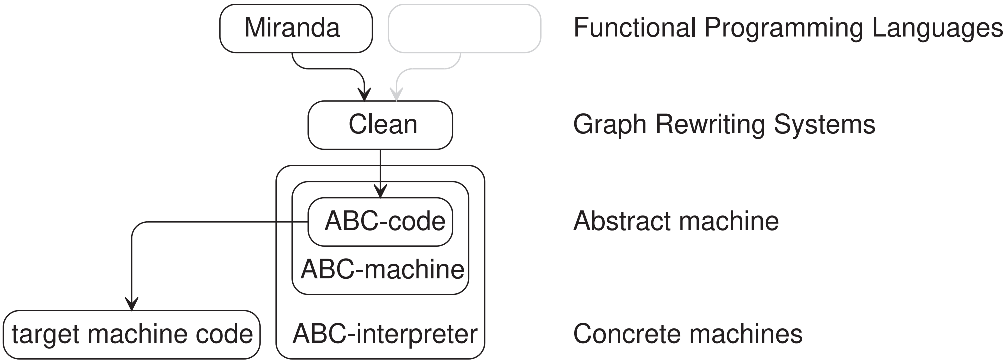


Figure 2. The place of the ABC-machine in the translation process of functional languages.

The first step on this translating path consists of the translation of functional languages to the graph rewriting language Clean[5,8]. In order to describe pattern matching and sharing of computations well this intermediate language is based on graph rewriting systems. An intermediate language based on the λ-calculus has problems to express these aspects and recursion. In this translation step the syntactical sugar of functional languages is removed. The sharing of computations is specified accurately. In many cases the translations from a functional program to an equivalent Clean program is straight forward. In other cases, like ZF-expressions, the transformations becomes rather complex. These transformations have also been formally specified in a functional language[9]. In the next step Clean is translated to ABC-code. This translation links the world of graphs and strategies to a sequence of imperative instructions. Finally, ABC-instructions are translated to concrete machine code. During this translation all details of the concrete machine become important to achieve an efficient implementation. A simple implementation of the ABC-machine is relatively easy to make, for instance using macro expansion.

There are several reasons to introduce the additional intermediate level defined by the abstract machine on the compilation path. First of all, the abstract machine helps to get a more structured implementation. When the abstraction is properly originated, irrelevant machine dependent issues are omitted. The trade-offs in the imperative graph rewriting process are shown clearly. Some of the restrictions imposed by a concrete machine, such as limited resources (like the limited stack size) are not necessarily present in an abstract machine. Of course, all problems have to be solved on one level or the other. However, the separation of concerns helps to get a more structured view on the problems (and their solutions) one has to face when making an implementation of functional languages.

Another advantage of the additional intermediate level is that the portability of the implementation is increased. To implement the functional languages on a new concrete machine only a new ABC-machine has to be implemented. Since the ABC-machine is an imperative machine on a relative low level of abstraction it is much easier to implement than Clean, or a high level functional language directly.

4.1 Graph rewriting in Clean

In order to understand the architecture of the ABC-machine and code generation it is important to have a proper view of the operational behaviour of a Clean program. A complete description of Clean is outside the scope of this work, we refer to[5,6,10]. The semantics of Clean will be explained informally below.

Clean is a lazy higher order functional programming language based on Functional Graph Rewriting Systems, it is designed as intermediate language between arbitrary functional languages and sequential machines. A **Functional Graph Rewriting System** (*FGRS*) is a Graph Rewriting System using the functional reduction strategy[10]. A reduction strategy is a function indicating which redex has to be rewritten. The functional strategy prescribes the same evaluation order as used in Miranda; lazy evaluation. A

Graph Rewriting System (*GRS*) is an extension of Term Rewriting Systems where the terms are replaced by directed graphs in order to avoid the duplication of work via sharing of expressions[6]. A **Term Rewriting System** (*TRS*) is a computational paradigm consisting of a collection of rewrite rules to transform terms (expressions) into equivalent terms[11].

A Clean program consists of a set of (typed) graph rewrite rules. The type system is based on the Milner/Mycroft type inference scheme[12,13].

A subgraph is a **redex** (*red*ucable *ex*pression) if there is a **left hand side** (*lhs* or *pattern*) of rewrite rule that matches this graph. A match is a mapping from the pattern to the graph that is the identity on constants and preserve the node structure. A graph is in **root normal form** (*rnf*) if the whole graph is not a redex and will never become a redex. A graph is in **normal form** (*nf*) if it does not contain any redex.

The rewrite rules are used to reduce the initial graph containing the symbol Start to normal form. The functional reduction strategy is used: rewrite alternatives are tried in textual order; patterns are matched from left to right; evaluation to root normal form is forced before an actual argument is compared with non-variable part of the pattern. It is possible to deviate from the functional strategy by adding strictness annotations, denoted by !, to the rewrite rules. Such a graph is reduced eagerly to root normal form.

A redex is rewritten by constructing the graph specified in the **right hand side** (*rhs*) of the rule: the **contractum**. Then all references to the root redex are **redirected** to the root of the contractum. There are also rewrite rule alternatives consisting only of a redirection; no contractum is specified in these rule. Nodes that cannot be reached from the root of the graph are **garbage**, they must be removed from the graph.

A small example is used to illustrate graph reduction in Clean. The example is even so small that no sharing of computations occurs.

```
::  Start -> INT                           ;  ||  The type of the start rule
    Start -> Length 0 (Cons 3 (Cons 4 Nil)) ;  ||  The rewrite rule for Start

::  Length !INT !(List x)  -> INT           ;  ||  Both arguments  of Length are strict
    Length n (Cons a b)    -> Length (+ n 1) b |  ||  + is the delta rule to add integers
    Length n Nil           -> n             ;  ||  2nd alternative; a redirection
```

The reduction process is illustrated by the following rewriting sequence (the redex rewritten is underlined), it is depicted below. The graphs correspond to each of the steps shown. The garbage produced in one rewrite step is drawn gray and removed in the next snapshot. The dataroot is usually not shown, it is included here to show the redirections clearly.

```
    Start                                  ||  a:  the only redex; apply the Start rule
→       Length 0 (Cons 3 (Cons 4 Nil))     ||  b:  this graph as a whole is the new redex
→       Length ( + 0 1 ) (Cons 4 Nil)      ||  c:  the strict arguments are reduced first
→       Length 1 (Cons 4 Nil)              ||  d:  rewrite according to first alternative
→       Length ( + 1 1 ) Nil               ||  e:  again one strict argument to be reduced
→       Length 2 Nil                       ||  f:  1st alternative does not match; use 2nd
→       2                                  ||  g:  this graph is in normal form
```

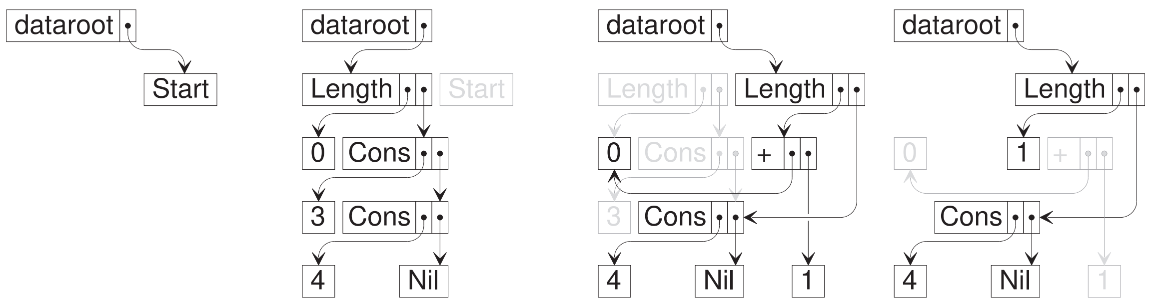
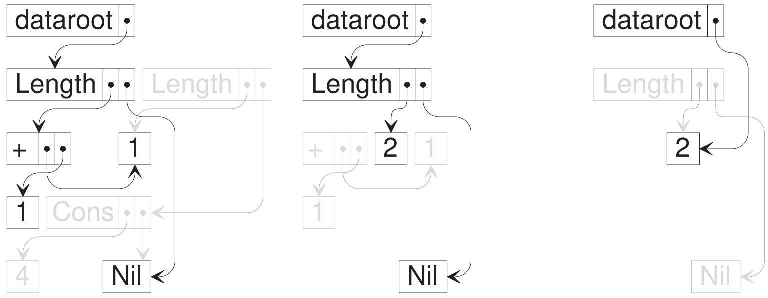


Figure 4 a b c d



4.2 Overview of the ABC-machine

The main part of the instruction set of the ABC-machine is designed to conveniently express graph rewriting. When this instruction set is mapped on a traditional concrete machine the resulting code will be executed relatively slowly, since present-day architectures are not at all designed for graph rewriting. A graph structure is not directly available on these architectures, but it has to be represented by some data structure. The modification of the graph will lead to complex memory management problems, involving garbage collection. For many simple calculations the use of graphs is not effective. This observation has triggered the introduction of the second part of the abstract machine. This part is an abstraction from a traditional stack-based architecture. To obtain an efficient program these fast instructions are used and graph rewriting is avoided whenever possible!

The ABC-machine consists of the following components:

- the *A(rgument)-stack* used to indicate the actual arguments in the graph store;
- the *B(asic value)-stack* used to hold and manipulate basic values efficiently;
- the *C(ontrol)-stack* to remember return addresses;
- the *graph store* containing the graph which is rewritten;
- the *descriptor store* containing information about symbols in the rewrite system;
- the *program counter* indicating the instruction to be executed, this is an address in the program store;
- the *program store* containing the instruction sequence to be executed;
- an *i(nput)-o(utput) channel* to enable interaction with the world.

This machine can be depicted as shown in figure 3.

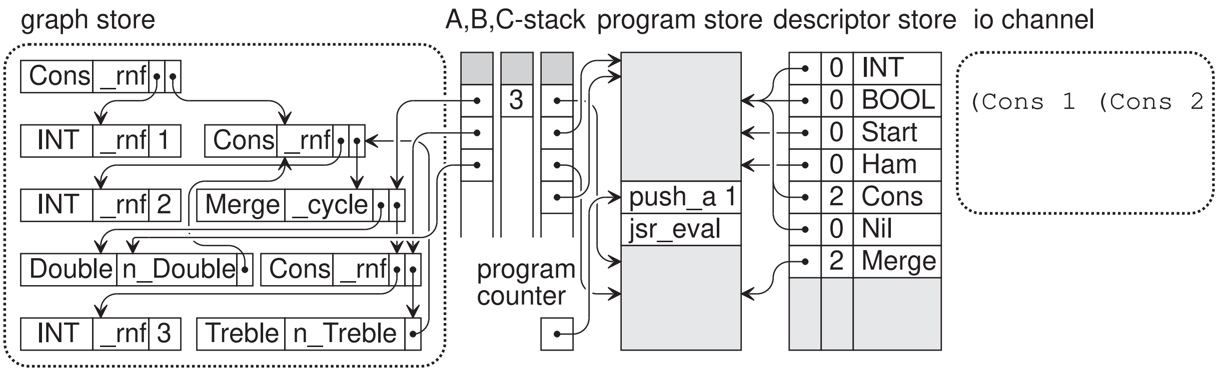


Figure 3. The ABC-machine.

4.3 The state of the ABC-machine

All components mentioned in the description above are part of the state of the ABCmachine. For convience the parts forming the traditional memory are grouped together in the tuple memory. The order of components is chosen to be similar to the components of Mac-1 discussed above as much as possible.

```
state    ==   (pc, astack, bstack, cstack, memory)
memory   ==   (graphstore, descstore, programstore, io)
```

4.4 The micro-instruction set of the ABC-machine

The components of the ABC-machine are abstract data-types. They are sufficiently specified by the operations defined on them. The Miranda implementation of the micro-instructions is not presented here. These Miranda definitions specify the semantics of the micro-instructions, on the other hand they suggest too much a specific implementation.

The following type synonyms are used to increase the clarity of the type definitions. The type nat stands for natural numbers including zero, it is represented by Miranda numbers (num).

```
arity    == nat
a_dst    == nat    || the index of the destination on the A-stack
a_src    == nat    || the index of the source on the A-stack
b_dst    == nat    || the index of the destination on the B-stack
b_src    == nat    || the index of the source on the B-stack
c_src    == nat    || the index of the source on the C-stack
nr_args  == nat    || the number of arguments involved
arg_nr   == nat    || the number of the argument involved
```

The complete micro-instruction set of the ABC-machine is listed in appendix 2. In this section only an overview of the kind of micro-instructions available is give.

The program counter

Since the ABC-machine has an imperative nature it has a locus of control; the **program counter**. The program counter contains the instr-id of the next instruction to be executed.

There are micro-instructions provided to initiate the program counter (points to the first instruction of the program) and to change it.

The A-stack

The **A-stack** contains node-id's; references to nodes stored in the graph store. It is used to access the actual arguments and the result of the rewrite rule executed. The top of the stack has index 0 (as is the case for the other stacks).

The B-stack

If the calculation of a simple numerical value would be done by building nodes, performing redirections and the like, it is obvious that an efficient implementation cannot be achieved on traditional hardware that has no support for these kind of actions whatsoever. On a traditional machine simple calculations are performed on a stack using only a couple of simple instructions. In order to obtain the desired behaviour for these calculations, the ABC-machine is equipped with a stack to hold basic values.

The **B-stack** contains basic values like integers and booleans. Such values are stored untagged on the B-stack. This means that it is impossible to determine the type of the elements of the B-stack.

Beside the updating micro-instructions there are micro-instructions defined to perform computations with the basic values stored on the B-stack. Usually, the arguments are on top of the B-stack and are replaced by the result of the operation. A typical example is:

```
bs_addi :: bstack -> bstack
```

The C-stack

The control stack (**C-stack**) is used to implement nested subroutines on the abstract machine. The program counter can be stored and recovered from the C-stack.

The graph store

The graph in the **graph store** is basically a Clean-like graph. It is composed of **nodes**. Each node is labelled with a unique identification, a **node-id**.

Each node contains a **descriptor identification** (descid) instead of a symbol. This is an entry in the descriptor store indicating the symbol. Furthermore, a node contains a sequence of node-id's representing the arguments. Instead of a descid and arguments a node can also contain a basic value like an integer or boolean. To reduce the number of definitions only integers are considered, so booleans, characters, reals and strings are not treated here.

The graph in the graph store will be examined by the program which tries to reduce the graph to normal form. For this purpose it is convenient that additional information is stored in the nodes of the graph. A node contains a **context**; the instr-id (instruction identification) referring to the first instruction of an instruction sequence. This ABC-instruction sequence will reduce the corresponding node to root normal form when it is executed. The instr-id stored in a node can be changed during reduction for markings of the node. This is used to determine at run-time that a node is already under reduction.

The micro-instructions show that the graph store deals with variable sized nodes. Overwriting a previously created node with new values is always possible.

The descriptor store

The ABC-machine is a piece of memory where symbol descriptors are stored. This **descriptor store** contains information about the symbols used in the rewrite system. Given the descriptor identification, **descid**, a descriptor can be taken from the descriptor store. This store can be initiated by passing a list of descriptors to the ds_init micro instruction.

The descriptor contains information of the associated symbol; its arity, the start address of the reduction code and the name of the symbol. Information can be retrieved from the descriptor by the corresponding micro-instructions.

The program store

The ABC-machine contains a sequence of machine instructions representing the reduction algorithm: the **program**. This program will rewrite the initial graph to its normal form according to the annotated functional strategy (see below). In contradiction to Mac-1 the elements from this store are only used as instruction, this implies that it is not needed nor wanted to store the instructions in some encoded form. In the Mac-1 each memory word can be interpreted as eithre a number or an instruction.

Each instruction has a unique identification: the **instr-id**, in order to be indicated as the current instruction by the program counter. A program cannot change during execution, it is loaded in the machine when the machine is booted.

The input-output channel

The abstract machine furthermore contains an **input-output** channel used to describe the result of the reduction visible for the outside world. On the output channel strings can be printed. These strings are appended to the existing output channel.

4.5 The instruction set of the ABC-machine

Each machine instruction of the ABC-machine is defined in terms of the micro-instructions. An instruction consists of an instruction identifier and zero or more operands. Not all instructions are shown in this section. The specification in terms of micro-instructions is only given for the most important instructions. For some other instructions only the type and an informal explanation is given. A precise definition of all machine instructions is given in[2,7].

The instructions are classified according to their main purpose:

- graph manipulation;

- retrieving information from a node;
- manipulating the A-stack;
- manipulating the B-stack;
- changing the flow of control;
- generating output.

Graph manipulation

There is only one instruction to create a new node in the graph store. Several instructions can be used to change the contents of existing nodes. All instructions for graph manipulation (with exception of the instruction create) have as operand an offset in the A-stack, to find the node-id of the node to manipulate.

The instruction create creates a new empty node in the graph store, the node-id of the new node is pushed on the A-stack.

```
create (pc, as, bs, cs, (gs, ds, ps, io))
 = (pc, as', bs, cs, (gs', ds, ps, io))
   where as'              = as_push nodeid as
         (gs', nodeid)    = gs_newnode gs
```

The most elaborated instruction to update the contents of a node is fill. The arguments of the node are taken from the A-stack; the first argument is the top of the A-stack.

```
fill desc nr_args instrid a_dst (pc, as, bs, cs, (gs, ds, ps, io))
 = (pc, as', bs, cs, (gs', ds, ps, io))
   where as'     = as_popn nr_args as
         gs'     = gs_update nodeid (n_fill desc instrid args) gs
         nodeid  = as_get a_dst as
         args    = as_topn nr_args as
```

Other instructions to change the contents of an existing node are:

```
fill_a     :: a_src -> a_dst -> instruction    || the copy node instruction;
filli      :: int -> a_dst -> instruction      || fills the node with the given integer;
filli_b    :: b_src -> a_dst -> instruction    || fills  with the integer at given depth on B-stack;
set_entry  :: instrid -> a_dst -> instruction  || changes the reduction context of the node;
```

Building a node in the graph store always involves two instructions: a create, which leaves a node-id on the A-stack and one of the fill instructions.

Example: to create the graph Cons 1 Nil, the following instruction sequence can be used. Assume that "_rnf" indicates some ABC-instruction sequence, probably just containing a rtn instruction since the newly created nodes are already in root normal form. The descriptor-id's of Nil and Cons are indicated by "Nil" and "Cons". This code fragment is written in the ABC-assembly language introduced below.

```
[   Create                     ,  ||  node for Cons
    Create                     ,  ||  node for Nil; 2nd arg of Cons
    Fill    "Nil" 0 "_rnf" 0   ,  ||  fill node just created
    Create                     ,  ||  node for 1; 1st arg of Cons
    Filli   1 0                ,  ||  fill node just created with the integer 1
    Fill    "Cons" 2 "_rnf" 2  ]  ||  fill Cons node
```

Retrieving information from a node

The information retrieved form a node is pushed on one of the stacks or stored in the program counter. The node-id of the node is found at the indicated depth on the A-stack. There are also instructions to test whether the contents of a node has the given value.

```
push_args a_src arity nr_args (pc, as, bs, cs, mem)
 = (pc, as', bs, cs, mem)
   where as'     = as_pushn args as
         args    = n_nargs (gs_get nodeid (mem_gs mem)) nr_args arity
         nodeid  = as_get a_src as
```

```
pushi_a a_src (pc, as, bs, cs, mem)
 =  (pc, as, bs', cs, mem)
    where   bs'       =  bs_pushi int bs
            int       =  n_i (gs_get nodeid (mem_gs mem))
            nodeid    =  as_get a_src as

eqi_a int a_src (pc, as, bs, cs, mem)
 =  (pc, as, bs', cs, mem)
    where   bs'       =  bs_pushb equal bs
            equal     =  n_eq_i (gs_get nodeid (mem_gs mem)) int
            nodeid    =  as_get a_src as

eq_desc_arity descid arity a_src (pc, as, bs, cs, mem)
 =  (pc, as, bs', cs, mem)
    where   bs'       =  bs_pushb equal bs
            equal     =  (n_eq_descid node descid) & (n_eq_arity node arity)
            node      =  gs_get nodeid (mem_gs mem)
            nodeid    =  as_get a_src as
```

Manipulating the A-stack

The A-stack is used to access the nodes involved in a rewriting. Via push instructions new node-id's can be pushed on the stack. In addition the following instructions are provided to manipulate the A-stack:

```
pop_a nr_args (pc, as, bs, cs, mem)
 =  (pc, as', bs, cs, mem)
    where   as'       =  as_popn nr_args as

push_a a_src (pc, as, bs, cs, mem)
 =  (pc, as', bs, cs, mem)
    where   as'       =  as_push nodeid as
            nodeid    =  as_get a_src as

update_a a_src a_dst (pc, as, bs, cs, mem)
 =  (pc, as', bs, cs, mem)
    where   as'       =  as_update a_dst nodeid as
            nodeid    =  as_get a_src as
```

Manipulating the B-stack

For the B-stack the same stack handling instructions as for the A-stack are defined:

```
pop_b       ::  nr_args -> instruction
push_b      ::  b_src -> instruction
update_b    ::  b_src -> b_dst -> instruction
pushi       ::  int  -> instruction
```

There are also instructions to manipulate the basic values on this stack. A typical example is:

```
addi (pc, as, bs, cs, mem)
 =  (pc, as, bs', cs, mem)
    where   bs'       =  bs_addi bs
```

There are many more instructions to do arithmetic, they all follow the same scheme as the addi instruction presented here. They are not listed here for reasons of brevity.

Changing the flow of control

The control flow has to be realized by manipulating the program counter. These jumps are unconditional, or directed by the boolean value on top of the B-stack.

```
jmp address (pc, as, bs, cs, mem)
 =  (pc', as, bs, cs, mem)
    where    pc'        =  address
```

```
jmp_false address (pc, as, bs, cs, mem)
 =  (pc', as, bs', cs, mem)
    where    pc'        =  pc          ,  if bs_getb 0 bs
                        =  address  ,  otherwise
             bs'        =  bs_popn 1 bs
```

When a jsr (jump to subroutine) instruction is executed, the current value of the program counter is stored on the C-stack, a rtn instruction will restore the program counter and pop the return address from the C-stack.

```
jsr address (pc, as, bs, cs, mem)
 =  (pc', as, bs, cs', mem)
    where    pc'        =  address
             cs'        =  cs_push pc cs
```

```
rtn (pc, as, bs, cs, mem)
 =  (pc', as, bs, cs', mem)
    where    pc'        =  cs_get 0 cs
             cs'        =  cs_popn 1 cs
```

A jsr_eval instruction will start the execution of the code addressed in the node referenced by the top of the A-stack, the return address is saved on the C-stack. Hence, it performs a jsr to the address stored in the node. By convention, executing this instruction sequence will reduce the node to its root normal form.

```
jsr_eval (pc, as, bs, cs, mem)
 =  (pc', as, bs, cs', mem)
    where    pc'        =  n_entry (gs_get nodeid (mem_gs mem))
             nodeid     =  as_get 0 as
             cs'        =  cs_push pc cs
```

Generating output

To show the result of the reduction there are print instructions. These instructions append strings to the output channel.

```
print string (pc, as, bs, cs, (gs, ds, ps, io))
 =  (pc, as, bs, cs, (gs, ds, ps, io'))
    where    io'        =  io_print string io
```

```
print_symbol a_src (pc, as, bs, cs, (gs, ds, ps, io))
 =  (pc, as, bs, cs, (gs, ds, ps, io'))
    where    io'        =  io_print string io
             node       =  gs_get (as_get a_src as) gs
             string     =  symbol_to_string node desc
             desc       =  ds_get (n_descid node) d
```

4.6 Execution of ABC-programs

To run the ABC-machine described here, the instructions must be applied to the state of the machine. Once started the machine must continue to execute instructions until the halt instruction is executed. While the machine is running, the current instruction is fetched and applied to the present state continuously.

The instruction cycle

The instruction fetch cycle recursively fetches the current instruction out of the program and applies it on the current state. The fetching (and hence the machine) stops when the program counter indicates that a halt instruction is executed.

```
instruction_cycle :: state -> state
instruction_cycle (pc, as, bs, cs, mem)
 =  (pc, as, bs, cs, mem)                              ,  if pc_end pc
 =  instruction_cycle (currinstr (pc, as, bs, cs, mem))    ,  otherwise
    where   pc'         =  pc_next pc
            currinstr   =  ps_get pc (mem_ps mem)
```

Booting the machine

Before it is possible to run the machine, the machine is loaded with the initial state (booted). The program and descriptors must be supplied as argument to the boot function. All parts of the machine are initiated by the corresponding init micro-instructions. The machine starts evaluating the program on the first instruction.

```
boot :: ([instruction], [desc]) -> state
boot (program, descriptors)
 =  (pc, as, bs, cs, (gs, ds, ps, io))
    where   pc   =  pc_init
            as   =  as_init
            bs   =  bs_init
            cs   =  cs_init
            gs   =  gs_init
            ps   =  ps_init program
            io   =  io_init
            ds   =  ds_init descriptors
```

4.7 Graph rewriting on the ABC-machine

In this section an informal description of graph rewriting on the ABC-machine is given. The functional strategy and the rewrite algorithm are combined into a single piece of ABC-code for every rewrite rule[2,14]. Associated with each rewrite rule there is a sequence of ABC-instructions which reduce a node in the graph to its root normal form. Using the functional strategy a node must be reduced to rnf, as soon as rewriting is initiated.

A graph reduction program in the ABC-machine consists of:

- rule dependent code to reduce a redex to its rnf according to the functional strategy;
- code for the predefined rules (delta rules like + etc.);
- a **run-time system**: a fixed piece of code that initiates the reduction of the Start rule to normal form and prints the reduct.

For each Clean rule there is ABC-code to:

- prepare the arguments, i.e. construct a stack frame as expected by the code for the rule alternatives;
- match and rewrite according each rule alternative;
- handle the situation that none of the rule alternatives is applicable. Code according to an additional rule alternative is generated for this purpose.

These pieces of code and their **calling conventions** (the interface between caller and callee) will be illustrated below. Arguments and results can be passed among functions in the graph, on the A-stack and on the B-stack. It is more efficient to handle references to object on the A-stack instead of in nodes of the graph: the references are directly available on the stack. When appropriate it is even more efficient to pass objects on the B-stack: the objects themselves are handled instead of references to nodes containing the objects.

The overall assumption is that once initiated the reduction of a graph continues until it is in rnf. Each rule alternative expects an A-stack frame containing a reference to the redex currently reduced and the node-id's of all arguments. The reference to the first argument is on top of the stack.

ABC-assembler

References to instructions are made by the address of the instruction within the program. This is fine for the ABC-machine and close to reality in a concrete machine, but cumbersome to read and write for human beings. To make ABC-programs better understandable an assembler level is introduced allowing the uses of symbloic names. Assembler statements can be mapped directly to instructions, but the assembler uses labels instead of addresses. A program in ABC-assembler is represented by a Miranda data structure containing a constructor for each instruction and additional constructors to store labels and descriptors.

```
assembler   ==   [statement]
statement
 ::=   Label        label                              |
       Descriptor   desc_label red_label arity name    |
       Fill         label nr_args label a_dst          |
       Jmp          label                              |
       etc…
```

Translating Clean to ABC-Code

The translations are described by a set of functions transforming a data structure representing a Clean program to ABC-assembler introduced above. Such a data structure is called an **abstract syntax tree** (AST). An AST is much easier to handle than concrete syntax. Moreover, the type system of the description language guarantees that only proper syntax trees are possible. The translation from Clean to ABC-code is illustrated by an example, it is formally specified the functional programming language Miranda in[2,7].

The goal of the translation is, of course, to transform Clean programs to equivalent and efficient ABC-programs. Since ABC-code is translated to (or interpreted by) various concrete machines it is not possible assign execution times to the instructions, hence it is impossible to develop an optimum compilation scheme.

To increase the performance of the generated code some deviations of the operational semantics of Clean[8] are made. Although Clean graphs are mapped directly to graphs in the graph store of the ABC-machine this store is not updated after every rewrite step. During the rewriting the information is stored on the stacks and (implicit) in the instruction sequence executed. The performance of ABC-programs heavily manipulating the graph store for simple computations will be low compared to ABC-code that employ the B-stack whenever appropriate. One of the ways to use the B-stack is outlined.

Example of the rewriting process

As example the rewrite rule length, introduced above, is considered. The ABC-program generated by the specification[2] is given below. The B-stack is not used to pass arguments in this code.

```
[               Descriptor
"Length"        "a_Length" 2 "Length"      ,  || The descriptor for the symbol Length
                Label                         || The node entry: top of stack indicates redex
"n_Length",     Set_entry     "_cycle" 0   ,  || Mark node to detect cyclic computations
                Push_args     0 2 2        ,  || Push the arguments on the A-stack
                Label                         || The apply entry: reduce the strict arguments
"a_Length",     Jsr_eval                   ,  || Reduce first argument to rnf
                Push_a        1            ,  || Copy second argument to top of stack
                Jsr_eval                   ,  || Reduce it to rnf
                Pop_a         1            ,  || Pop duplicated second argument
```

```
              Label                                ||  Entry for first rule alternative
"Length1",    Eq_desc_arity  "Cons" 2 1     ,      ||  Match second argument
              Jmp_false      "Length2"      ,      ||  Continue with next alternative if match fails
              Push_args      1 2 2          ,      ||  Push sub-arguments;
              Push_a         1              ,      ||  Rewrite according alternative 1; Push b
              Jsr_eval                      ,      ||  Reduce it
              Create                        ,      ||  Node for result of + n 1
              Create                        ,      ||  Node for 1
              Filli          1 0            ,      ||  Fill this node
              Push_a         5              ,      ||  Push n. It is known to be a rnf
              Jsr            "+1"           ,      ||  Reduction of + n 1
              Update_a       1 5            ,      ||  Adapt stack frame for call of Length
              Update_a       0 4            ,      ||
              Pop_a          4              ,      ||  Remove old (sub-)arguments
              Jmp            "Length1"      ,      ||  Continue with alternative 1 of Length
              Label                                ||  Entry for second rule alternative
"Length2",    Eq_desc_arity  "Nil" 0 1      ,      ||  Match argument
              Jmp_false      "Length3"      ,      ||  Continue with next alternative if match fails
              Fill_a         0 2            ,      ||  Rewrite according alternative 2; copy node
              Pop_a          2              ,      ||  Remove arguments from stack
              Rtn                           ,      ||  Rnf reached
              Label                                ||  Generated entry for additional alternative
"Length3",    Jmp            "type_error"   ]      ||  This must be a type error!
```

In the figures below several snapshots of the ABC-machine state are shown during the reduction of a graph according to the code for the rewrite rule Length. A node is drawn grey until its contents is determined by the ABC-program.

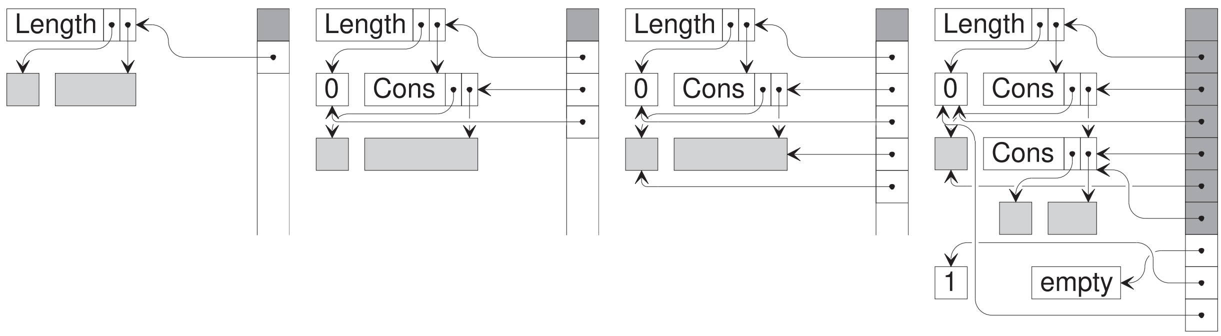
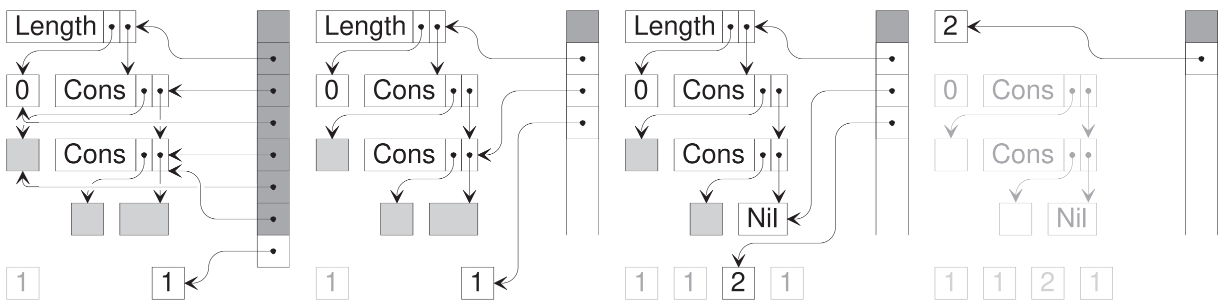


Figure 5 a **b** **c** **d**

a: Initial state at node entry. The state in this figure corresponds to Fig 4.b
b: State at first rewrite alternative entry.
c: After successful match of first rule alternative.
d: Reduction according the first rule alternative. First, the last strict argument is reduced. The state before calling +1 to reduce first strict argument is shown. The state corresponds to Fig 4.c.



e **f** **g** **h**

e: State returned by +1. Corresponds to Fig 4.d.
f: State before calling Length1 recursively. State corresponding to Fig 4.e.
g: State before last call of Length1. Corresponds to Fig 4.f.
h: State of ABC-machine after reaching the rnf. Corresponds to Fig 4.g.

Optimizations

The code generated by the basic compilation scheme can be improved at many points. The most important improvement is to use the B-stack instead of nodes to pass basic values between functions. The use of the B-stack is described informally and illustrated with an example.

The extended compilation scheme using the B-stack

Basic values are manipulated always on the B-stack in the ABC-machine. Every computation involving basic values requires the transportation of the values from the graph to the B-stack and the shipment of the result back to a node in the graph store. When the result is used again as argument in another computation there is much data transportation. To reduce the unnecessary movement of data, the compilation scheme must be changed such that basic values stay on the B-stack as much as possible.

To achieve this, the calling conventions for the rewrite alternatives are changed: strict arguments of a basic type are passed on the B-stack and the result of a reduction is left on the B-stack when it is of a basic type. The complexity of the code generation is increased significantly by this new calling convention. Arguments and results must be moved to the desired place at every occurrence. This is not difficult, but involves an elaborate case analysis.

An example of the use of the B-stack

To show the effect of using the B-stack snapshots of the reduction process corresponding to the pictures above are shown. Both arguments are strict. The first argument is also a basic value. So, it will be passed on the B-stack instead of as an address in the graph on the A-stack. The result will also be passed on the B-stack by the rule alternatives. The rewrite entry for + also expects its arguments and leaves its result on the B-stack.

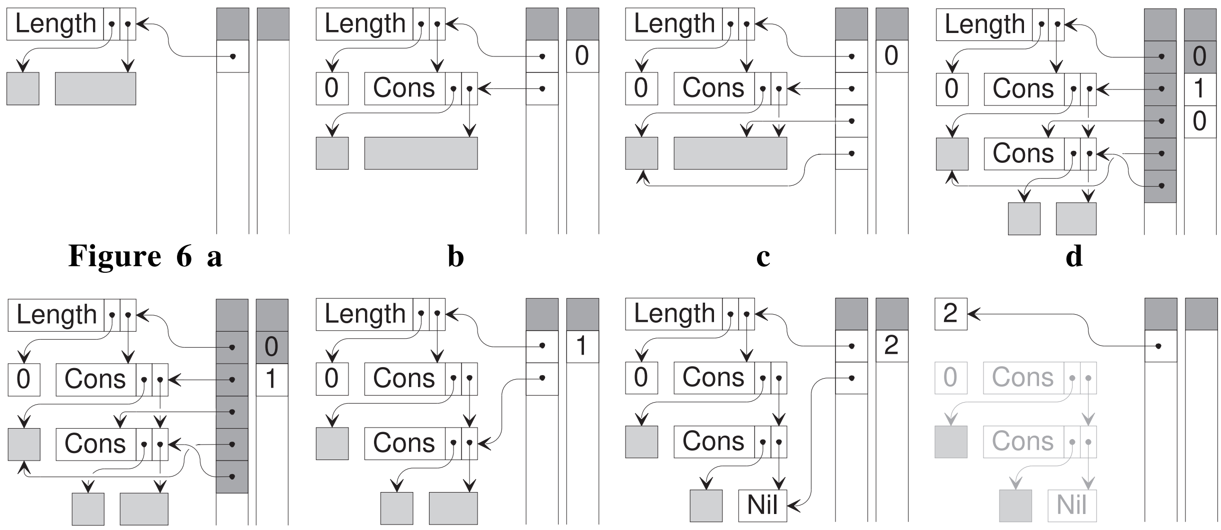


Figure 6 a **b** **c** **d**

e **f** **g** **h**

The efficiency gained by this optimized compilation scheme depends on the rules to transform and the implementation of the ABC-machine. The efficiency gain for the length rule given as example using a reasonable implementation[15] is a factor 2.5. Moreover, a lot of garbage collection is prevented. The efficiency gained by prohibiting the garbage collection depends heavily on the circumstances, we have measured an additional garbage collection effort of four times the execution time of the optimized length function.

There are several implementations of the ABC-machine. The specification of the abstract ABC-machine appeared to be a very useful and a valuable definition of the abstract machine.The implementation in C of a parallel ABC-interpreter[15] follows the specification presented here very closely. A state-of-the-art speed is achieved by the translation of ABC-code to MC68000 code for a SUN3/280[16] and Mac II[17]. The number of function calls per second is given by the nfib-number, it is obtained by dividing the reduct of the following program by the execution time.

```
Start      ->  Nfib 32                                            ;
Nfib n     ->  If (< n 2) 1 (++ (+ (Nfib (-- n)) (Nfib (- n 2)))   ;
```

This number is just over 1.000.000 on a Mac IIfx showing that the implementation approch followed here was indeed very successful. **WAT IS HET IN C ??? ****

5 Discussion

This paper shows that functional languages can be used with success as a description formalism for (abstract) machines.

The description of the instruction set of Mac-1 in Miranda is a bit longer than the description given in the original table 1. Nevertheless, a very clear and compact specification is obtained by using this description method. Due to the two level description the operational semantics of each instruction is clearly specified in terms of simple micro-instructions. The micro-instructions are simple access function defining the machine components.

The use of a functional programming language as description formalism has a number of advantages:

- The description can be verified by the compiler for the functional language and has well defined semantics.
- The specification is more complete. For instance, in the functional description of Mac-1 it is clear when the program counter is incremented, and that the incremented program counter is stored on the stack in a call instruction.
- The description serves as a prototype implementation of the machine and is able to execute programs. When programs execute correctly on the prototype, this increases the convidence in the correctness of the description and that the specification given is the one intended.
- The description is more direct. There are no temporary variables needed, as used in the imperative description of the swap instruction. This is an advantages of having an explicit state in the description of the instructions instead of an implicit global state.

It is a pity that Miranda lacks graph rewrite semantics, this limits the use of the specification to trace the execution of programs.

The abstract machine specification of the ABC-machine presented here has successfully been used in a large software project as prototype implementation and it appeared to be an useful definition for the actual implementation. It appeared to be easy to add some instructions and to experiment with slightly different instruction sets. The development of this specification was forcing a well structured design and the development time was spent on designing the machine instead of solving problems with its specification.

It appears to be valuable that the specification of compilation scheme to translate Clean to ABC-assembler is executable; the generated code can be observed and executed. Execution serves as a test for the specification. Due to its complex nature it is very hard to verify its correctness otherwise. The prototype has been used to execute Clean programs. Important observations of the dynamically behaviour can and have be obtained in this way. Using these prototypes it is easy to glance at the code generated and to observe the consequences of small changes made in the ABC-machine specification.

Implementations of Clean, incorperating the ABC-machine for Apple Macintosh, Sun 3 and Sun 4 and the complete specification of the ABC-machine can be obtained by anonymeous ftp from clean@cs.kun.nl (131.174.33.1).

References

1 D.A. Turner, Miranda: A non-strict functional language with polymorphic types. Proc. of the conference on Functional Programming Languages and Computer Architecture, *Springer LNCS* **201**. 1 - 16. 1985.

2 P.W.M. Koopman, *Functional programs as executable specifications*. Ph. D. Thesis, University of Nijmegen, The Netherlands. 1990.

3 A.S. Tanenbaum, *Structured computer organization*, 2nd ed Prentice-Hall 1984

4 H.P. Barendregt *The lambda-calculus, Its Syntax and Semantics* (revised edition). Studies in logic and foundations of mathematics. **103** North-Holland, Amsterdam. 1984.

5 T. Brus, M.C.J.D. van Eekelen, M. van Leer and M.J. Plasmeijer, Clean - A Language for Functional Graph Rewriting. Proc. of the Third International Conf. for Functional Prog. Languages and Comp. Architectures (FPCA '87), Portland, Oregon, USA. *Springer LNCS* **274**, 364-384. 1987.
6 H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer and M.R. Sleep, Towards an Intermediate Language based on Graph Rewriting. Proc. of Parallel Architectures and Languages Europe (PARLE), part II, Eindhoven, The Netherlands. *Springer LNCS.* **259**. 159-175. 1987.
7 M.J. Plasmeijer and M.C.J.D. van Eekelen, *Functional Programming and Parallel Graph Rewriting*. To appear in 1993 by Addison Wesley. ISBN 0201416638.
8 M. C. J. D. van Eekelen, E. G. J. M. H. Nöcker, M. J. Plasmeijer and J. E. W. Smetsers, *Concurrent Clean*. Technical report 89-18, October 1989. University of Nijmegen. The Netherlands.
9 P.W.M. Koopman and E.G.J.M.H. Nöcker, *Compiling Functional Languages to Term Graph Rewrite Systems*. Internal Report 88-1, Department of Computer Science, University of Nijmegen, The Netherlands. 1988
10 M.C.J.D. van Eekelen, *Parallel Graph Rewriting, Some Contributions to its Theory, its Implementation and its Application*. Ph. D. Thesis, University of Nijmegen, the Netherlands. 1988.
11 J.W. Klop, *Term Rewriting systems: a tutorial*. Centre for Mathematics and Computer Science, CWI Amsterdam, The Netherlands. Note CS-N8701. 1987.
12 R.A. Milner, Theory of Type Polymorphism in Programming. *Journal of Computer System Sciences*, Vol **17**, no 3. 348 - 375. 1978.
13 A. Mycroft, Polymorphic type schemes and recursive definitions. Proc. of the 6th Int Conf on Programming. *Springer LNCS* **167**. 217 - 228. 1984.
14 J.E.W. Smetsers, *Compiling Clean to Abstract ABC-Machine Code*. Technical Report 89-20. Department of Computer Science, University of Nijmegen, the Netherlands.1989.
15 E.G.J.M.H. Nöcker, *The PABC Simulator v0.5. Implementation Manual*. Technical report no. 89-19, October 1989. Department of Informatics. University of Nijmegen. The Netherlands.
16 G.A. Weijers, *ABC-machine implementation Guide*, Technical report University of Nijmegen. To appear in 1990.
17 J.C. van Groningen. *Implementing the ABC-machine on MC680X0 based architectures.* M. Sc. thesis. University of Nijmegen. November 1990.
?? P.W.M. Koopman bla bla CSN'92.
?? J.M. Spivey. *The Z notation: A Reference Manual*, Hemel Hempstead, Prentice Hall, 1989.
?? Th. Johnson. *Compiling lazy functional programming languages*. Dissertation at Chalmers University, Götenborg, Sweden, 1987.
?? D.A. Schmidt. *Denotational Semantics, a methodology for language development*. Allyn and Bacon. Boston. 1986.
?? M.J.C. Gordon. *The denotational Description of Programming Languages, An Introduction.* Springer-Verlag, New York, 1979.

Appendix 1: The micro-instructions of the Mac-1 machine

The micro-instruction level of the Mac-1 machine is constructed on top of the abstract types mem and word. The signature of these types and an implementation is given in this appendix.

```
abstype mem
   with  get      ::  address -> mem -> word
         update   ::  address -> word -> mem -> mem
         store    ::  program -> mem
```

To stay close to a concrete machine model the memory is modelled as a list of words with length 4096.

```
mem  ==    [word]

get address mem             =  mem ! addressTOnum address
update address word mem     =  take n mem ++ (word : drop (n+1) mem) where n = addressTOnum address
store program               =  take (2^12) [instrTOword instr | instr <- program] ++ repeat zero)


abstype  word
   with  add                   ::  word -> word -> word
```

```
sub                  ::  word -> word -> word
is_neg               ::  word -> bool
eq                   ::  word -> word -> bool
wordTOinstruction    ::  word -> instruction
instrTOword          ::  instr -> word
addressTOnum         ::  address -> num
wordTOnum            ::  word -> num
numTOword            ::  num -> num -> word
```

```
add x y     =  (x + y) mod 2^16                 || 16-bit signed addition
sub x y     =  (x - y) mod 2^16                 || 16-bit signed subtraction
is_neg x    =  x >= 2^15
eq x y      =  x=y
wordTOnum w     =  cond (is_neg w) (w-2^16) w
numTOword n     =  n          , if 0 <= n < 2^15
                =  n+2^16     , if -2^15 <= n < 0
addressTOnum w  =  w mod 2^12                   || bit 16 .. 13 are ignored

zero  =  numTOword 0
one   =  numTOword 1
```

The conditional function `cond` used above is defined as:

```
cond :: bool -> * -> * -> *
cond c then else   =  then   , if c
                   =  else   , otherwise

wordTOinstruction word
 =  (True, lodd  x  ),  if word div 2^12 = 0          ||  word = `0000 x'
 =  (True, stod  x  ),  if word div 2^12 = 1          ||  word = `0001 x'
 =  (True, addd  x  ),  if word div 2^12 = 2          ||  word = `0010 x'
 =  (True, subd  x  ),  if word div 2^12 = 3          ||  word = `0011 x'
 =  (True, jpos  x  ),  if word div 2^12 = 4          ||  word = `0100 x'
 =  (True, jzer  x  ),  if word div 2^12 = 5          ||  word = `0101 x'
 =  (True, jump  x  ),  if word div 2^12 = 6          ||  word = `0110 x'
 =  (True, loco  x  ),  if word div 2^12 = 7          ||  word = `0111 x'
 =  (True, lodl  x  ),  if word div 2^12 = 8          ||  word = `1000 x'
 =  (True, stol  x  ),  if word div 2^12 = 9          ||  word = `1001 x'
 =  (True, addl  x  ),  if word div 2^12 = 10         ||  word = `1010 x'
 =  (True, subl  x  ),  if word div 2^12 = 11         ||  word = `1011 x'
 =  (True, jneg  x  ),  if word div 2^12 = 12         ||  word = `1100 x'
 =  (True, jnze  x  ),  if word div 2^12 = 13         ||  word = `1101 x'
 =  (True, call  x  ),  if word div 2^12 = 14         ||  word = `1110 x'
 =  (True, pshi     ),  if word = 15*2^12+ 0*2^8      ||  word = `1111 0000 0000 0000'
 =  (True, popi     ),  if word = 15*2^12+ 2*2^8      ||  word = `1111 0010 0000 0000'
 =  (True, push     ),  if word = 15*2^12+ 4*2^8      ||  word = `1111 0100 0000 0000'
 =  (True, pop      ),  if word = 15*2^12+ 6*2^8      ||  word = `1111 0110 0000 0000'
 =  (True, retn     ),  if word = 15*2^12+ 8*2^8      ||  word = `1111 1000 0000 0000'
 =  (True, swap     ),  if word = 15*2^12+10*2^8      ||  word = `1111 1010 0000 0000'
 =  (True, insp  y  ),  if word div 2^8 = 252         ||  word = `1111 1100 y'
 =  (True, desp  y  ),  if word div 2^8 = 253         ||  word = `1111 1101 y'
 =  (False, undef   ),  otherwise
    where    x  =  numTOword (word mod 2^12)
             y  =  numTOword (word mod 2^8)

instrTOword (LODD   x)  =   0*2^12 +  x        , if 0 <= x < 2^12   ||  word = `0000 x'
instrTOword (STOD   x)  =   1*2^12 +  x        , if 0 <= x < 2^12   ||  word = `0001 x'
instrTOword (ADDD   x)  =   2*2^12 +  x        , if 0 <= x < 2^12   ||  word = `0010 x'
instrTOword (SUBD   x)  =   3*2^12 +  x        , if 0 <= x < 2^12   ||  word = `0011 x'
instrTOword (JPOS   x)  =   4*2^12 +  x        , if 0 <= x < 2^12   ||  word = `0100 x'
instrTOword (JZER   x)  =   5*2^12 +  x        , if 0 <= x < 2^12   ||  word = `0101 x'
```

```
instrTOword (JUMP   x)  =   6*2^12 +  x            , if 0 <= x < 2^12   ||  word = `0110 x'
instrTOword (LOCO   x)  =   7*2^12 +  x            , if 0 <= x < 2^12   ||  word = `0111 x'
instrTOword (LODL   x)  =   8*2^12 +  x            , if 0 <= x < 2^12   ||  word = `1000 x'
instrTOword (STOL   x)  =   9*2^12 +  x            , if 0 <= x < 2^12   ||  word = `1001 x'
instrTOword (ADDL   x)  =  10*2^12 +  x            , if 0 <= x < 2^12   ||  word = `1010 x'
instrTOword (SUBL   x)  =  11*2^12 +  x            , if 0 <= x < 2^12   ||  word = `1011 x'
instrTOword (JNEG   x)  =  12*2^12 +  x            , if 0 <= x < 2^12   ||  word = `1100 x'
instrTOword (JNZE   x)  =  13*2^12 +  x            , if 0 <= x < 2^12   ||  word = `1101 x'
instrTOword (CALL   x)  =  14*2^12 +  x            , if 0 <= x < 2^12   ||  word = `1110 x'
instrTOword  PSHI       =  15*2^12 +  0*2^8                             ||  word = `1111 0000 0000 0000'
instrTOword  POPI       =  15*2^12 +  2*2^8                             ||  word = `1111 0010 0000 0000'
instrTOword  PUSH       =  15*2^12 +  4*2^8                             ||  word = `1111 0100 0000 0000'
instrTOword  POP        =  15*2^12 +  6*2^8                             ||  word = `1111 0110 0000 0000'
instrTOword  RETN       =  15*2^12 +  8*2^8                             ||  word = `1111 1000 0000 0000'
instrTOword  SWAP       =  15*2^12 + 10*2^8                             ||  word = `1111 1010 0000 0000'
instrTOword  STOP       =  15*2^12 + 10*2^8 + 1                         ||  word = `1111 1010 0000 0001'
instrTOword (INSP   y)  =  15*2^12 + 12*2^8 + y  , if 0 <= y < 2^8      ||  word = `1111 1100 y'
instrTOword (DESP   y)  =  15*2^12 + 13*2^8 + y  , if 0 <= y < 2^8      ||  word = `1111 1101 y'
instrTOword (CONS   c)  =  c                     , if 0 <= c < 2 ^ 15
                        =  c + 2^16              , if -2^15 <= c < 0
```

Appendix 2: The micro-instructions of the ABC-machine

This appendix contains the signature of the abstract data types specifiying the components of the abstract ABC-machine.

```
abstype  pc, astack, bstack, cstack, graphstore, descstore, programstore, io
with
```

The program counter

```
pc_init      ::  instrid
pc_next      ::  instrid -> instrid
pc_halt      ::  instrid -> instrid
pc_end       ::  instrid -> boolean
```

The A-stack

```
as_init      ::  astack
as_get       ::  a_src -> astack -> nodeid
as_topn      ::  nr_args -> astack -> nodeid_seq
as_popn      ::  nr_args -> astack -> astack
as_push      ::  nodeid -> astack -> astack
as_pushn     ::  nodeid_seq -> astack -> astack
as_update    ::  a_dst -> nodeid -> astack -> astack
```

The B-stack

```
bs_init      ::  bstack
bs_get       ::  b_src -> bstack -> basic
bs_geti      ::  b_src -> bstack -> int
bs_popn      ::  b_src -> bstack -> bstack
bs_push      ::  basic -> bstack -> bstack
bs_pushb     ::  boolean -> bstack -> bstack
bs_pushi     ::  int -> bstack -> bstack
bs_pushn     ::  basic_seq -> bstack -> bstack
bs_update    ::  b_dst -> basic -> bstack -> bstack
bs_addi      ::  bstack -> bstack
```

The C-stack

```
cs_init       :: cstack
cs_get        :: c_src -> cstack -> instrid
cs_popn       :: nr_args -> cstack -> cstack
cs_push       :: instrid -> cstack -> cstack
```

The graph store

```
gs_get        :: nodeid -> graphstore -> node
gs_init       :: graphstore
gs_newnode    :: graphstore -> (graphstore,nodeid)
gs_update     :: nodeid -> (node -> node) -> graphstore -> graphstore

n_arg         :: node -> arg_nr -> arity -> nodeid
n_args        :: node -> arity -> nodeid_seq
n_arity       :: node -> arity
n_descid      :: node -> descid
n_entry       :: node -> instrid
n_i           :: node -> int
n_nargs       :: node -> nr_args -> arity -> nodeid_seq
n_eq_arity    :: node -> arity -> boolean
n_eq_descid   :: node -> descid  -> boolean
n_eq_i        :: node -> int  -> boolean
n_eq_symbol   :: node -> node  -> boolean
n_copy        :: node ->  node -> node
n_fill        :: descid -> instrid -> args ->  node -> node
n_filli       :: descid -> instrid -> int ->  node -> node
n_setentry    :: instrid -> node -> node
```

The descriptor store

```
ds_get        :: descid -> descstore -> desc
ds_init       :: [desc] -> descstore

d_ap_entry    :: desc -> instrid
d_arity       :: desc -> arity
d_name        :: desc -> string
```

The program store

```
ps_get        :: instrid -> programstore -> instruction
ps_init       :: [instruction] -> programstore
```

The iput-output channel

```
io_init       :: io
io_print      :: string -> io -> io
```