Chapter 10
The abstract ABC machine

This chapter introduces the architecture and instruction set of an *abstract* machine: the *ABC machine* (Koopman *et al.*, 1990). The instruction set of the ABC machine is used as an intermediate language in the compilation path from functional language to concrete machine instructions (Figure 10.1). This intermediate language is on a much more concrete level of abstraction than Clean. With this abstract machine graph reduction can be described in terms of low-level, imperative machine instructions.

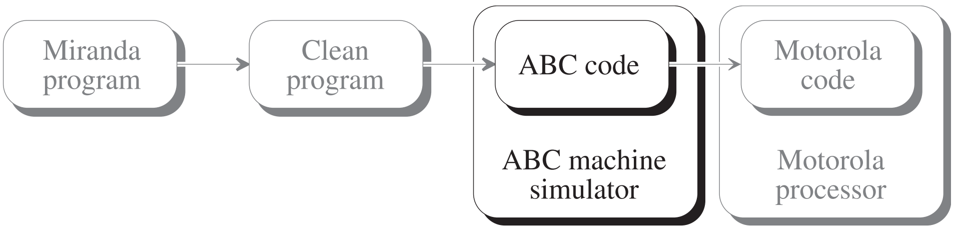

Figure 10.1 Graph rewriting on a more concrete level: the ABC machine and its instruction set.

First a motivation and global overview of the architecture of the ABC machine is given (Section 10.1). A, B and C stand for the three stacks of this abstract machine. The architecture of the ABC machine is a mixture between an idealized graph rewriting machine and a more traditional stack-based machine architecture.

The ABC machine is specified in a functional language, so in this chapter we also show that a very elegant specification of a highly imperative system can be given in a functional language. In this way a prototype implementation of the abstract machine is obtained, almost effortlessly, that can actually be used to execute ABC instructions. A complete specification of the ABC machine is found in Appendix C.

The ABC machine is composed of several kinds of *memory storage*. The contents of all these storages together define the *state* of the machine. Each storage is defined as an abstract data structure in the description. Access functions defined on these abstract data structures form the *micro-instructions* of the machine. The storage components of the ABC machine and the available micro-instructions are explained in Section 10.2. The actual *ABC machine instructions* are defined with the help of these micro-instructions in Section 10.3. The remaining parts of the machine (the *instruction fetch cycle* and a *bootstrapper*) are defined in Section 10.4. In order to allow the use of symbolic names one can add an assembler to convert ABC assembly statements into proper ABC instructions.

Chapter 11 discusses in detail how Clean programs can be compiled into ABC code. The translation of ABC code into concrete machine code is discussed in Chapter 12.

10.1 About the ABC machine

The additional intermediate level created with the abstract ABC machine is very important since it helps to get a more structured implementation. By specifying how an FGRS-based language like Clean can be implemented on an abstract machine, a general blueprint is available for actual implementations on a large class of concrete target architectures. When the abstraction is carefully chosen, irrelevant machine aspects can be left out. For instance, an unbounded amount of memory or an infinite number of processors can be assumed to be available in the abstract machine. Furthermore, one can abstract from addressing modes, word sizes, register use and other particularities. This makes it easier to understand how FGRSs can conceptually be implemented and what the trade-offs are.

Another advantage of the additional intermediate level is that a *compiler* structured in this way can be ported more easily: one only has to change the last part of such a compiler that translates ABC machine code into concrete machine code (see Chapter 12). The loss of efficiency introduced by this additional intermediate level is in most cases very small.

The definition of an abstract machine also makes it easy to achieve a machine-independent *interpreter* for functional languages, e.g. by implementing an ABC machine *simulator* in a portable high-level language. Such a simulator has to interpret ABC machine instructions.

Another well-known abstract machine for functional languages is the G machine (Johnsson, 1984). The objectives of this abstract machine are identical to those of the ABC machine: defining an imperative abstract graph rewriting machine as an intermediate level in the compilation of functional languages. Basically, the G machine and the ABC machine have much in common, although they differ in almost all details. The most important differences on the architecture level are caused by the different representation of nodes: ABC nodes have variable arity while the nodes in the G-machine are either leaf nodes or application nodes with two arguments. Furthermore, the ABC machine is designed to handle arbitrary constructors.

10.1.1 The basic architecture of the ABC machine

The architecture of the abstract ABC machine actually consists of a mixture of two abstract machines.

One part of the ABC machine consists of an idealized graph rewriting architecture with a graph storage designed to perform graph rewriting conveniently. However, it is not easy to map this part of the abstract machine efficiently on a traditional concrete machine. The present-day concrete machine architectures are not at all designed for graph rewriting. They do not have anything like a graph store. A graph on a concrete machine has to be represented by some data structure stored in a linear memory of the machine. The rewriting of the graph will lead to complex memory management problems at run-time, involving garbage collection.

When simple calculations are realized on a concrete machine by building graphs and performing redirections it must be clear that this will be terribly inefficient compared with the way computations are commonly performed on such a computer: registers and stacks are normally used where possible. This observation has triggered the introduction of the second part of the abstract ABC machine: an abstraction from a traditional stack-based architecture.

So the architecture of the ABC machine is a mixture between the idealized graph rewriting machine one would like to have in reality and an idealized concrete machine.

The most important optimization that results in an efficient implementation of functional languages is: use the instructions of the traditional part of the abstract machine where possible. These instructions have the property that they can relatively easily be mapped on a concrete architecture. Avoid the higher level graph rewriting instructions of the abstract machine because they are generally hard to implement efficiently on a concrete machine.

The **ABC machine** consists of the following memory components (Figure 10.2):

- a *graph store* containing the graph to be rewritten;
- a *program store* containing the instruction sequence to be executed;

- a *program counter* containing an identification of the instruction (instrid) to be executed;
- an *A(rgument)-stack* to reference nodes in the graph store;
- a *B(asic value)-stack* to deal with basic values efficiently;
- a *C(ontrol)-stack* to store and retrieve return addresses;
- a *descriptor store* containing information about the symbols used;
- an *Input)/O(utput) channel* to show the result of a function.

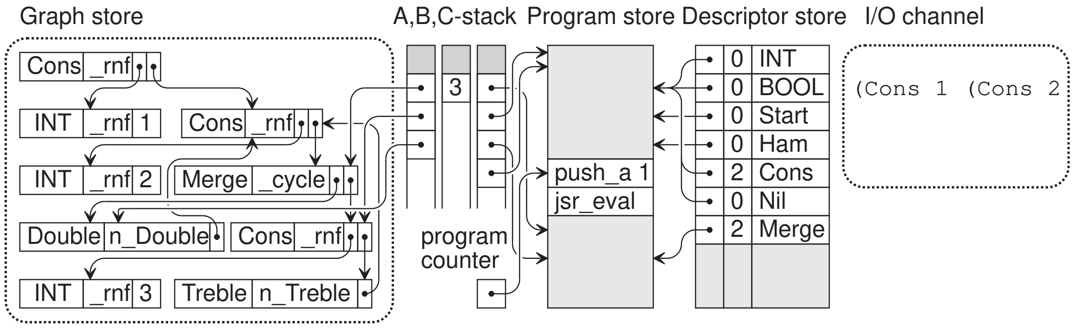

Figure 10.2 A snapshot of a possible state of the ABC machine (see Section 10.2). Not all pointers are drawn in order to keep the picture illustrative. The symbol names and labels in the nodes are actually pointers into the descriptor store and the program store.

The idealized and inefficient graph rewriting part of the ABC machine consists of the graph store, A-stack and descriptor store. The more traditional and efficient part of the abstract machine consists of the B-stack. The other, also conventional, components of the machine are shared by both parts: the program store, the program counter, the C-stack and the I/O channel.

10.1.2 The description method

The ABC machine is described in a functional language. Such a description has a number of advantages over conventional descriptions:

- Using the abstraction mechanisms of the language, like abstract data types, a hierarchical description can be given. A layered machine description enables a very clear description: details can be described without getting an overall view cluttered by too many details at the top level of the specification.
- The compiler can partially check the specification. This does not imply that the description is correct, but frequently occurring errors, like forgotten arguments, type conflicts and unbound identifiers, will be spotted by the compiler.

- The machine specification is its own prototype implementation, since it is a functional program. This implies that at an early stage the machine can be run to observe its dynamic behaviour.

A formal description of a representative subset (only Boolean and integer values are considered) of the ABC machine is given in Appendix C. The description method used below to define the ABC machine is also well suited for the description of arbitrary (concrete) machine architectures (Koopman *et al.*, 1993; Koopman, 1990).

Micro-instructions

The bottom layer of the description consists of the specification of the different kinds of memory storage that are visible to the programmer. Each memory component of the ABC machine, the A-stack (as), B-stack (bs), C-stack (cs), graph store (gs), descriptor store (ds), program counter (pc), program store (ps) and I/O channel (io), is described by an abstract data type with corresponding access operations. These access functions are called **micro-instructions** (see Section 10.2).

The state of the machine

The **state** of the machine is defined by the contents of all memory components. This state is described by a tuple:

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

Machine instructions

Instructions change the state of the machine, e.g. they update one or more memory components. Hence, **ABC instructions** are functions that take the current machine state as argument and deliver a new state, changed by the instruction. So all instructions yield a function of the following type:

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

Memories can only be accessed using the micro-instructions of the machine. This means that ABC instructions are always described in terms of these micro-instructions. Each description of an ABC instruction shows how information flows between the different memory components when the instruction is executed (see Section 10.3).

Execution of the ABC machine

The specification of the ABC instructions and micro-instructions can be regarded as part of the interpretation circuit of the abstract machine. When a *bootstrapper* and a circuit to *fetch instructions* are added, a complete executable ABC machine is obtained (see Section 10.4).

10.2 Machine components and micro-instructions

In this section the rationale behind the different machine components with the available micro-instructions is given. The abstract data types for the memory storage of the ABC machine are assumed to be sufficiently defined by the specification of the abstract operations that can be applied to them. For further details the reader is referred to the concrete operational semantics of the micro-instructions given in Appendix C.

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

```
arity   == nat   || the arity of a symbol
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
```

10.2.1 The program store

A Clean program is translated into a *sequence of ABC instructions*, the **ABC program**, which is stored in the **program store**. Each instruction has a unique identification. An instruction identification (an **instr-id**) is like an ordinary machine address. The instructions will rewrite the initial graph to its normal form according to the annotated functional strategy. Conceptually there are two algorithms involved, the *annotated functional reduction strategy* which indicates the next redex and the *rewriting* of that redex according to the Clean rules. These algorithms are merged in the ABC program to increase efficiency.

Each Clean rule alternative is translated into a sequence of ABC instructions. Such an instruction sequence controls the reduction order, checks whether the indicated subgraph matches the rule, and if so, builds a contractum in the graph store to which references to the root of the corresponding redex are redirected. If the rule alternative does not match the subgraph, the instruction sequence corresponding to the next rule alternative is executed. The execution of an instruction sequence corresponding to a Clean rule alternative does not always have to start with the same instruction. Depending on the circumstances under which a function is called (e.g. curried or not) a particular instruction **entry** (address of the first instruction of an instruction sequence) is chosen.

A program does not change during execution. It is loaded once in the machine (ps_init) when the machine is booted. Programs to be stored are denoted by a list of instructions (see Section 10.4). An instruction that has to be executed is fetched from the program store (ps_get).

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

10.2.2 The program counter

Since the ABC machine has an imperative nature it is essential to have a locus of control: the **program counter**. The program counter is an abstract data structure containing the *instr-id* of the next instruction to be executed. A micro-instruction (pc_init) is provided to initialize the program counter such that it points to the first instruction of the program. The program counter can be incremented: the next instruction in the sequence will be executed (pc_next), the counter can be set to indicate that the program is finished (pc_halt), or it can be checked whether the last instruction of the program is reached (pc_end). The instr-id can be fetched from (pc_get) and assigned to (pc_update) the program counter.

```
pc_init   :: pc
pc_next   :: pc -> pc
pc_halt   :: pc -> pc
pc_end    :: pc -> bool
pc_get    :: pc -> instrid
pc_update :: instrid -> pc -> pc
```

10.2.3 The graph store

The ABC machine has, of course, a piece of memory in which the graph to be rewritten is stored: the **graph store**. As in Chapter 5, each node in the graph has a unique identification, the **node-id**. There are micro-instructions to generate a new (empty) graph store (gs_init), to create a new (empty) node in the graph (gs_newnode) and to retrieve information stored in a node of the graph (gs_get). Finally, a node can be updated by a function passed as a parameter to the gs_update micro-instruction.

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

The nodes in the graph store

A node in the graph store does not contain a *symbol*, but instead it contains a *descriptor identification* (**descr-id**) which is an **entry** (address where the information can be found) in the descriptor store (see Section 10.2.4). The advantage of this is that the (possibly long) symbol name need not be stored in a node. The actual name of a symbol is only needed when results have to be printed. During reduction it is only important that all symbols have a unique (short) representation such that the equality of symbols can be tested easily. This is accomplished by having a unique entry for each symbol in the descriptor table.

A node generally also contains a *sequence of node-ids* representing the arguments of the symbol. An exception is formed by arguments that are basic values, such as integers or booleans. For efficiency they are stored directly in the nodes themselves. This is in contrast with the way such graphs are structured in Chapter 5.

Each node furthermore contains a **code field** in which an instruction entry is stored. By convention, the execution of the corresponding sequence of ABC instructions will take care of the reduction of the node to root normal form. The contents of the code field can be *changed* at run-time, so it can be used as a kind of flag. Different kinds of code can be executed under different kinds of circumstances. For instance, it might be used at run-time to determine that a node is already under reduction. This means that there is a cycle of nodes each of which requires the evaluation of all the others in order to reach a root normal form: a **cycle-in-spine error**.

There are several micro-instructions to extract information stored in a node. The nodes themselves can be extracted from the graph store by the micro-instruction gs_get defined above.

```
n_arg      :: node -> arg_nr -> arity -> nodeid
n_args     :: node -> arity -> nodeid_seq
n_arity    :: node -> arity
n_B        :: node -> bool
n_descrid  :: node -> descrid
n_entry    :: node -> instrid
n_I        :: node -> int
n_nargs    :: node -> nr_args -> arity -> nodeid_seq
```

There are also micro-instructions to test whether a piece of a node has a certain value. They are important for pattern matching.

```
n_eq_arity     :: node -> arity -> bool
n_eq_B         :: node -> bool -> bool
n_eq_descrid   :: node -> descrid -> bool
n_eq_I         :: node -> int -> bool
n_eq_symbol    :: node -> node -> bool
```

The following micro-instructions to change the contents of a node are passed as argument to gs_update:

```
n_copy      :: node -> node -> node
n_fill      :: descrid -> instrid -> nodeid_seq -> node -> node
n_fillB     :: descrid -> instrid -> bool -> node -> node
n_fillI     :: descrid -> instrid -> int -> node -> node
n_setentry  :: instrid -> node -> node
```

In order to build a contractum new nodes have to be created in the graph store and redirections have to be performed. Redirections are conceptually elegant and explain the semantics of graph rewriting very well. However, it is very inefficient to examine the whole graph to substitute every reference to the root of a redex by a reference to the root of the contractum. Instead, it is actually much more efficient to *overwrite* the root node of the redex with the root of the contractum. The result is that all original references are automatically redirected to the contractum. The micro-instructions above show that the graph store can deal with variable sized nodes. So overwriting a node with another value is always possible. The update instruction is used both for overwriting old nodes as well as for the creation of new ones. New nodes are built as follows. First, a new empty node is created in the graph store (using gs_newnode). Then, the empty node is filled with the update instruction. To create an empty node first, which is filled later, is in particular convenient for the construction of cyclic structures (see Section 10.3).

10.2.4 The descriptor store

The ABC machine contains a piece of memory in which symbol descriptors are stored. This **descriptor store** contains information about the symbols used in the rewrite system. For each symbol and each basic type there is a unique entry in the descriptor store, labelled with the **descr-id**. Given a descriptor identification, a descriptor can be taken from the descriptor store (ds_get). This store can be initialized by passing a list of descriptors to the ds_init micro-instruction.

```
ds_get  :: descrid -> descrstore -> descr
ds_init :: [descr] -> descrstore
```

The descriptors

A **descriptor** contains information of the associated symbol: its *arity*, an instruction entry that is called for curried applications (see Section 11.2) and the *name of the symbol*. The symbol name is used only to print a representation of the graph in root normal form on the output channel. Information can be retrieved from a descriptor using the following micro-instructions:

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

10.2.5 The A-stack

The **A-stack** contains node-ids: references to nodes stored in the graph store. It is used to access the actual arguments and the result of the applied rewrite rule. As in imperative languages, instead of the data structure itself, a reference to it is passed to or returned from a function. The top of the stack has index 0 (as is the case for the other stacks). A new, empty, A-stack can be created by:

```
as_init      :: astack
```

An element at any depth, or a sequence of nr_args top elements, can be retrieved from the A-stack by:

```
as_get       :: a_src -> astack -> nodeid
as_topn      :: nr_args -> astack -> nodeid_seq
```

The type nodeid_seq represents a sequence of node-ids which can be taken from or pushed onto the A-stack, and it serves also as the argument sequence for a node. The A-stack can be updated by:

```
as_popn      :: nr_args -> astack -> astack
as_push      :: nodeid -> astack -> astack
as_pushn     :: nodeid_seq -> astack -> astack
as_update    :: a_dst -> nodeid -> astack -> astack
```

10.2.6 The B-stack

The ABC machine also has a stack to hold basic values (such as integers, reals and Boolean values). Basic values are stored and computed on this **B-stack** instead of in nodes in the graph store whenever this is possible. Temporary values are also held on this stack. All values are stored untagged on the stack. The initial, empty, B-stack is created by:

```
bs_init      :: bstack
```

Information can be obtained from the B-stack with:

```
bs_get       :: b_src -> bstack -> basic
```

```
bs_getB    :: b_src -> bstack -> bool
bs_getI    :: b_src -> bstack -> int
```

The B-stack can be changed with:

```
bs_copy    :: b_src -> bstack -> bstack
bs_popn    :: nr_args -> bstack -> bstack
bs_push    :: basic -> bstack -> bstack
bs_pushB   :: bool -> bstack -> bstack
bs_pushI   :: int -> bstack -> bstack
bs_update  :: b_dst -> basic -> bstack -> bstack
```

Besides these updating micro-instructions there are micro-instructions defined to perform computations with the basic values stored on the B-stack. Usually, the arguments are all on top of the B-stack and are replaced by the result of the operation. When arguments are not on the B-stack they are an argument of the micro-instruction. Some micro-instructions to handle integer values are:

```
bs_addI    :: bstack -> bstack
bs_eqI     :: bstack -> bstack
bs_eqIi    :: int -> b_src -> bstack -> bstack
bs_gtI     :: bstack -> bstack
```

A further optimization would be to include *registers* in the ABC machine to hold these basic values. However, the number of registers and the operations possible on them vary too much between different concrete machines. Register allocation is therefore left as a part of the implementation of the ABC machine (see Chapter 12).

10.2.7 The C-stack

The **C-stack** (control stack) is used to implement nested reductions in the abstract machine. It contains return addresses (instr-ids). The program counter can be stored and recovered from this stack.

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

10.2.8 The input/output channel

The abstract machine furthermore contains an **input/output channel**. In the description in this chapter only simple *output* is considered, such that the result of the reduction can be shown. A string is printed just by appending it to the output channel. This corresponds to classical character-based output.

```
io_init          :: io
io_print         :: string -> io -> io
io_print_symbol  :: node -> descr -> io -> io
```

With respect to more sophisticated input/output using modern facilities such as windows, menus and dialogs the available interface on the machine (e.g. Macintosh tool box or X Window System interface) is assumed to be called directly as a side-effect. This is possible due to the properties of UNQ types (see Chapter 8).

10.3 The machine instructions

The state of the ABC machine is completely determined by the contents of its eight stores: the A-stack, B-stack, C-stack, the graph store, descriptor store, program counter, program store and input/output channel. Each ABC instruction changes the state of the machine by changing the contents of one or more of these stores. The only way to change the contents or retrieve information from the stores is via the micro-instructions introduced in the previous section. So the *ABC instructions* are described in terms of *micro-instructions* and for each instruction we define how the information flows between these memories. In this way *imaginary data paths* are defined between the several stores. One can imagine that these data paths could actually be present in a concrete piece of hardware.

Not all ABC instructions are shown in this section. The specification in terms of micro-instructions is only given for the most important ones. For some other instructions, only the type and an informal explanation are given. The instructions shown here 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.

10.3.1 Instructions for graph manipulation

There are several instructions to manipulate the graph store. With the instruction create a new empty node is created in the graph store. With other instructions the contents of an existing (possibly empty) node can be changed. Finally, there are also instructions that fetch information stored in the nodes of the graph. All instructions for graph manipulation (with the exception of the instruction create) have as operand an offset into 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, and the node-id of the new node is pushed on the A-stack. The create instruction is defined as:

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

With the fill instruction one can update the contents of an existing node:

```
fill:: descrid -> nr_args -> instrid -> a_dst -> instruction
fill descr nr_args entry a_dst (as,bs,cs,gs,ds,pc,ps,io)
   = (as',bs,cs,gs',ds,pc,ps,io)
     where  as'      =  as_popn nr_args as
            gs'      =  gs_update nodeid (n_fill descr entry args) gs
            nodeid   =  as_get a_dst as
            args     =  as_topn nr_args as
```

As can be seen from this specification the arguments of the node are taken from the A-stack. The first argument is on top of the A-stack. Other instructions to change the contents of an existing node are:

```
fill_a      ::  a_src -> a_dst -> instruction     ||  the copy node instruction
fillB       ::  bool -> a_dst -> instruction      ||  fills the node with the given bool
fillB_b     ::  b_src -> a_dst -> instruction     ||  fills with bool found on B-stack
fillI       ::  int -> a_dst -> instruction       ||  fills the node with the integer
fillI_b     ::  b_src -> a_dst -> instruction     ||  fills with integer found on B-stack
set_entry   ::  instrid -> a_dst -> instruction   ||  changes code entry of node
```

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

To create the graph Cons 1 Nil, the following code fragment is written in the ABC assembly language introduced in Section 10.4.

```
[   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
Fill "Cons" 2 "_rnf" 2         ]  || fill Cons node
```

It is assumed that calling the entry stored in the code field of a node will reduce the corresponding subgraph to root normal form. For subgraphs already in root normal form the instruction sequence can be one simple return instruction (rtn) assumed to be labelled by the entry _rnf. The descriptor-ids of Nil and Cons are indicated by "Nil" and "Cons".

Retrieving information from a node

The information retrieved from a node is generally stored on one of the stacks. The node-id of the node to fetch the information from is found at the indicated depth on the A-stack. There are also instructions to test whether the content of a node has a certain value.

```
push_args:: a_src -> arity -> nr_args -> instruction
push_args a_src arity nr_args (as,bs,cs,gs,ds,pc,ps,io)
  = (as',bs,cs,gs,ds,pc,ps,io)
    where as'     = as_pushn args as
          args    = n_nargs (gs_get nodeid gs) nr_args arity
          nodeid  = as_get a_src as

pushI_a:: a_src -> instruction
pushI_a a_src (as,bs,cs,gs,ds,pc,ps,io)
  = (as,bs',cs,gs,ds,pc,ps,io)
    where bs'     = bs_pushI int bs
          int     = n_I (gs_get nodeid gs)
          nodeid  = as_get a_src as

eqI_a:: int -> a_src -> instruction
eqI_a int a_src (as,bs,cs,gs,ds,pc,ps,io)
  = (as,bs',cs,gs,ds,pc,ps,io)
    where bs'     = bs_pushB equal bs
          equal   = n_eq_I (gs_get nodeid gs) int
          nodeid  = as_get a_src as

eq_descr_arity:: descrid -> arity -> a_src -> instruction
eq_descr_arity descrid arity a_src (as,bs,cs,gs,ds,pc,ps,io)
  = (as,bs',cs,gs,ds,pc,ps,io)
    where bs'     = bs_pushB equal bs
          equal   = n_eq_descrid node descrid & n_eq_arity node arity
          node    = gs_get nodeid gs
          nodeid  = as_get a_src as
```

Appendix C lists some more instructions for retrieving information from a node, such as extracting the arguments of a node or changing them.

10.3.2 Instructions to manipulate the A-stack

The A-stack is used to access the nodes involved in a rewriting. Via push instructions, new node-ids can be pushed on the stack. Furthermore, there are instructions provided to manipulate the A-stack:

```
pop_a:: nr_args -> instruction
pop_a n (as,bs,cs,gs,ds,pc,ps,io)
   =  (as',bs,cs,gs,ds,pc,ps,io)
      where   as'      =  as_popn n as

push_a:: a_src -> instruction
push_a a_src (as,bs,cs,gs,ds,pc,ps,io)
   =  (as',bs,cs,gs,ds,pc,ps,io)
      where   as'      =  as_push nodeid as
              nodeid   =  as_get a_src as

update_a:: a_src -> a_dst -> instruction
update_a a_src a_dst (as,bs,cs,gs,ds,pc,ps,io)
   =  (as',bs,cs,gs,ds,pc,ps,io)
      where   as'      =  as_update a_dst nodeid as
              nodeid   =  as_get a_src as
```

The cyclic graph ones: Cons 1 ones is constructed as follows (here the advantages of separate create and fill instructions are seen).

```
[    Create                     ,  || node for Cons
     Push_a 0                   ,  || 2nd arg of Cons
     Create                     ,  || node for 1; 1st arg of Cons
     FillI 1 0                  ,  || fill node just created
     Fill "Cons" 2 "_rnf" 2     ]  || fill Cons node
```

10.3.3 Instructions to manipulate the B-stack

The B-stack has stack handling instructions similar to the A-stack:

```
pop_b      ::  nr_args -> instruction
push_b     ::  b_src -> instruction
pushI      ::  int -> instruction
pushB      ::  bool -> instruction
update_b   ::  b_src -> b_dst -> instruction
```

There are many instructions to manipulate the basic values on this stack; they all follow the same scheme as the add instruction presented below.

```
addI:: instruction
addI (as,bs,cs,gs,ds,pc,ps,io)
```

```
   = (as,bs',cs,gs,ds,pc,ps,io)
     where  bs'      = bs_addI bs
```

10.3.4 Instructions to change the flow of control

The desired flow of control has to be realized by manipulating the program counter. Jumps can be unconditional but they can also be controlled by the Boolean value on top of the B-stack.

```
jmp:: instrid -> instruction
jmp address (as,bs,cs,gs,ds,pc,ps,io)
   = (as,bs,cs,gs,ds,pc',ps,io)
where   pc'      = pc_update address pc

jmp_false:: instrid -> instruction
jmp_false address (as,bs,cs,gs,ds,pc,ps,io)
   = (as,bs',cs,gs,ds,pc',ps,io)
     where  pc'      = pc, if bool
                     = pc_update address pc, otherwise
            bool     = bs_getB 0 bs
            bs'      = bs_popn 1 bs

jmp_true:: instrid -> instruction
jmp_true address (as,bs,cs,gs,ds,pc,ps,io)
   = (as,bs',cs,gs,ds,pc',ps,io)
     where  pc'      = pc_update address pc, if bool
                     = pc, otherwise
            bool     = bs_getB 0 bs
            bs'      = bs_popn 1 bs
```

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

```
jsr:: instrid -> instruction
jsr address (as,bs,cs,gs,ds,pc,ps,io)
   = (as,bs,cs',gs,ds,pc',ps,io)
     where  pc'      = pc_update address pc
            cs'      = cs_push (pc_get pc) cs

rtn:: instruction
rtn (as,bs,cs,gs,ds,pc,ps,io)
   = (as,bs,cs',gs,ds,pc',ps,io)
     where  pc'      = pc_update (cs_get 0 cs) pc
            cs'      = cs_popn 1 cs
```

The jsr_eval instruction is a very important instruction since it starts the execution at the entry stored in the code field of the node on top of the A-stack. The return address is saved on the C-stack. In other words, the jsr_eval performs a jsr to the instr-id stored in the node. By convention, the execution of the instruction sequence will reduce the corresponding node to its root normal form.

```
jsr_eval:: instruction
jsr_eval (as,bs,cs,gs,ds,pc,ps,io)
   =  (as,bs,cs',gs,ds,pc',ps,io)
      where  pc'      =  pc_update (n_entry (gs_get nodeid gs)) pc
             nodeid   =  as_get 0 as
             cs'      =  cs_push (pc_get pc) cs
```

The program execution stops after the execution of a halt instruction. The instruction fetch_cycle will spot this instruction and the machine will stop (see Section 10.4).

```
halt:: instruction
halt (as,bs,cs,gs,ds,pc,ps,io)
   =  (as,bs,cs,gs,ds,pc',ps,io)
      where  pc'      =  pc_halt pc
```

10.3.5 Instructions to generate output

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

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

print_symbol:: a_src -> instruction
print_symbol a_src (as,bs,cs,gs,ds,pc,ps,io)
   =  (as,bs,cs,gs,ds,pc,ps,io')
      where  io'      =  io_print_symbol node descr io
             nodeid   =  as_get a_src as
             node     =  gs_get nodeid gs
             descr    =  ds_get (n_descrid node) ds
```

10.4 Program execution

As already explained in Section 10.1 the specification of the ABC instructions and micro-instructions can be seen as part of the interpretation circuit of the abstract machine. In order to obtain an executable specification of the ABC machine that can be used to execute ABC programs one needs an implementation of the abstract data structures which are discussed in Section 10.2. Such an implementation is given in Appendix C. Still there are some machine parts missing. An *instruction fetching* mechanism has to be defined and one needs a *bootstrapper* to initialize the machine. This bootstrapper needs an ABC program with corresponding descriptor values in a machine-readable format. For this purpose a small *assembler* is defined that can convert an *ABC assembly* program to the desired format. The ABC assembly language makes it possible to denote ABC instructions in a user-friendly notation.

10.4.1 The instruction fetch cycle

To run the ABC machine described here, the instructions must be applied to the state of the machine. In each machine cycle recursively the current instruction is fetched from the program store and is applied to the current state. The fetching (and hence the machine) stops when the program counter indicates that a halt instruction is executed.

```
fetch_cycle :: state -> state
fetch_cycle (as,bs,cs,gs,ds,pc,ps,io)
   = (as,bs,cs,gs,ds,pc,ps,io)                              , if pc_end pc
   = fetch_cycle (currinstr (as,bs,cs,gs,ds,pc',ps,io))     , otherwise
     where  pc'       = pc_next pc
            currinstr = ps_get (pc_get pc) ps
```

10.4.2 Booting the machine

Before it is possible to run the machine, the machine must be loaded (booted) with the initial state. The program and descriptors must be supplied as arguments to the boot function. All parts of the machine are initialized by the corresponding _init micro-instructions.

```
boot :: ([instruction],[descr]) -> state
boot (program,descriptors)
   = (as,bs,cs,gs,ds,pc,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
```

10.4.3 ABC assembly language

For testing the machine specification with actual ABC programs it is convenient to have an ABC assembly language. For instance, in ABC instructions addresses (instr-ids) are actually required instead of symbolic labels. This is close to reality, but such abstract machine code is cumbersome to read and write for human beings. For this reason an assembly language is defined in which symbolic names are used instead of concrete machine addresses.

The ABC assembly language allows us to write, for example:

```
example1:: assembly
example1  =  [              Label
              "Length1",    Jsr_eval                              ,
                            Eq_descr_arity "Cons" 0 0             ,
                            Jmp_false "Length2"                   ,
                            Push_args 0 2 2                       ,
                            Create                                ,
                            Push_a 2                              ,
                            Fill "Length" 1 "n_Length" 1          ]
```

instead of the ABC instruction sequence that might correspond to it:

```
example2:: [instructions]
example2  =  [              jsr_eval                              ,
                            eq_descr_arity 23 0 0                 ,
                            jmp_false 62                          ,
                            push_args 0 2 2                       ,
                            create                                ,
                            push_a 2                              ,
                            fill 12 1 613 1                       ]
```

In this particular case, a program in ABC assembly language is represented by a Miranda data structure. Therefore, every ABC instruction is represented by a constructor with a similar name: the first character is changed to an upper case character to make it a constructor. Addresses are replaced by symbolic labels. Furthermore, there is a denotation for all objects involved. The use of a Miranda data structure has the advantage over the use of strings that it eliminates the generation and parsing of ABC assembly. It also enables easy printing and manipulating of ABC assembly statements (data structures) which is much harder for ABC machine instructions (functions).

An assembler translates an ABC assembly program into ABC instructions with the corresponding descriptors:

```
assembler:: assembly -> ([instruction], [descr])
```

10.4.4 Running a program

The ABC machine starts evaluating an assembly program as follows:

```
fetch_cycle (boot (assembler (assembly_program)))
```

Summary

- The ABC machine is an abstract machine in which two different kinds of architecture are combined:
  - one part of the abstract machine is an *idealized architecture* to perform *graph rewriting*;
  - the other part is an *abstraction* of a more or less *traditional stack-based machine architecture*.
- The ABC machine consists of the following components:
  - a *program store*, containing a sequence of instructions that controls the rewriting process;
  - a *program counter*;
  - a *graph store*, containing the graph to be rewritten;
  - an *argument stack*, containing node-ids;
  - a *basic value stack*, containing basic values;
  - a *control stack*, containing return addresses;
  - an *output channel*, to print the string representation of an obtained root normal form;
  - a *descriptor store*, containing the descriptors of function and constructor symbols.
- The instruction set of the machine offers another *machine-independent level of abstraction* for the implementation of functional languages.
- The imperative ABC machine can be specified elegantly in a functional language:
  - the *memory stores* are defined as abstract data types;
  - the machine has *micro-instructions*: operations defined on these abstract types to access the stores;
  - the *instructions* of the ABC machine are defined in terms of micro-instructions, thus defining the data flow between the several machine components;
  - an *assembly language* is defined that permits instruction labels and symbolic names to be used.
- The given specification of the ABC machine is executable. It can therefore be used as an interpreter of ABC instructions and, for

example with a Miranda to Clean compiler as an interpreter of Miranda scripts. In this way the behaviour of the ABC machine can be observed.

EXERCISES

10.1 Write small ABC assembly routines to accomplish the following tasks (the routines should remove their arguments and other created entries from the stacks with the exception of their result):

(a) A routine that takes two argument nodes containing integers on the A-stack, and leaves the sum of these integers on the B-stack.

(b) A routine that takes two integers on the B-stack and leaves only the greater of them on the B-stack (computes maximum).

(c) A routine that takes a boolean on the B-stack and builds a node containing the complement of this boolean (logical NOT).

10.2* Extend the machine with one register br that replaces the top of the B-stack. Define a suitable abstract data type for br. Give the definition and implementation of the following micro-instructions: br_getB, br_putB, br_getI, br_putI, br_init. Give the new definition of the ABC state. Give the new definition of pushB. Define new instructions that take advantage of the new micro-instructions.

10.3* Define the instruction set of a programmable pocket calculator using the description method introduced in this chapter. Define for such a machine: a program store and a stack. Define suitable micro-instructions on these machine components. Define the following instructions: addI, subI, mulI, divI, eqI, pop, push, label, jsr, rtn, jmp_false, start, stop. Define two instruction: one to clear the program store and one to add an instruction into the store. Program the machine to calculate the factorial function.