Chapter 12
Realizing the ABC machine

12.1 Representing the ABC machine components
12.2 Garbage collection
12.3 The Motorola MC68020 processor
12.4 Representing ABC components on the MC68020
12.5 Generating MC68020 code
12.6 Example of concrete code
12.7 Performance

This chapter treats the last step in the implementation path of functional languages: the realization of the abstract ABC machine by translation of ABC machine code to executable code for a concrete machine (Figure 12.1).

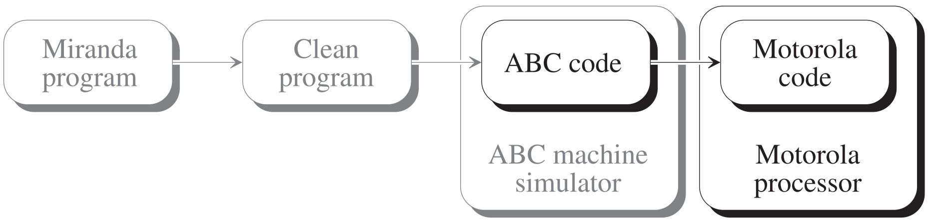

Figure 12.1 Realizing the ABC machine on a concrete machine.

In principle, an abstract machine such as the ABC machine can be realized either in hardware or by a software implementation running on standard hardware. With tailor-made hardware one should be able to obtain extremely good performance. However, the design of such a dedicated piece of hardware would take a lot of time and money. Furthermore, with present day compilation techniques it is possible to achieve efficient implementations on cheap standard hardware. So in practice there is no need for special purpose hardware reduction machines unless ultimate performance is demanded.

A software realization of the ABC machine can be achieved in two different ways. The first method is to write a program that interprets ABC machine code. Such an *interpreter* analyses and executes ABC code at run-time. In general, this is done instruction by instruction. Owing to the run-time analysis that has to be done, the most important disadvantage of an interpreter is that it is slow, independent of the language in which the interpreter is written. On the other hand, an interpreter has the advantage that it is easy to implement. Furthermore, it can be written in a portable language such as C (Kernighan and Ritchie, 1978) which makes an implementation available on many different kinds of machine. Tracing and debugging tools can also be encapsulated relatively easily into an interpreter. The formal definition of the ABC machine in Appendix C can be used as such an interpreter for ABC code.

The second method is to make a *code generator* that analyses sequences of ABC code at compile-time and generates executable code that does the actual work at run-time. In this way much time can be spent analysing the ABC code such that the code generated is as efficient as possible. Code generators are therefore generally much more complicated programs than interpreters. By making good use of the instructions and registers that are available on the concrete machine, code generators can produce efficient code that can do the work typically one order of magnitude faster than an interpreter running on the same machine. For ABC code sometimes an improvement of even two orders of magnitude has been measured (see Section 11.3). Concrete architectures differ much in detail (the instruction set, the memory management, number and kind of registers). Implementation tricks suited for one machine are usually not suited for others. So code generators that produce fast code are generally not very portable.

In this chapter we explain how to generate good code for standard hardware. First one has to decide how the logical memory stores of the ABC machine are mapped onto the available memory storage of the concrete machine. This is discussed in general terms in Section 12.1. Suitable concrete data structures have to be designed for this purpose. In general this will not cause many problems. The concrete data structures needed to implement the stores often directly follow the abstract definition in the ABC machine. An important difference between the ABC machine and a concrete machine is that the latter has a *limited* amount of memory. As a consequence, the memory management of the graph store becomes more complex. *Garbage collection* is needed to reclaim unused memory. Several techniques for garbage collection are discussed in Section 12.2.

Simple code generators (based on macro substitution of ABC instructions) are in principle not difficult to implement. But, when the context in which ABC instructions appear is taken into account, many optimizations can be performed. The ABC code generated is certainly not optimal in terms of efficiency. For example, many ABC instructions require their arguments on top of the stack whereas on a concrete machine the arguments of corresponding instructions can be accessed more directly using the addressing modes of the target machine. This implies that the copy actions specified by the ABC instructions are often not necessary. The ABC machine does not have registers. Registers have the property that their access time is much less than the access time of ordinary memory locations. Registers can be used for implementing the ABC stacks and heap, for storing temporary values, and for passing arguments and return values. To make optimal use of registers while avoiding unnecessary saving and restoring of information is a difficult task. Consequently, a good code generator for Clean is rather complex (Van Groningen, 1990). Some of the techniques that can be used are general techniques commonly used for the implementation of other types of languages (for example register allocation algorithms). Sometimes they have to be adapted because of the specific properties of both source language and target machine. Other optimizations are very specific for lazy functional languages, e.g. the implementation of higher order functions, efficient memory management and optimized graph reduction.

In the remaining sections of this chapter we describe how abstract ABC machine code can be translated to a specific target machine: the Motorola MC68020 processor. However, many of the ideas presented can be generally applied for other architectures as well. A brief summary of the Motorola architecture and instruction set is given in Section 12.3. A concrete representation of the ABC memory components chosen on the Motorola is presented in Section 12.4. How computations specified in ABC code can be optimized when generating concrete machine code is described in Section 12.5. A concrete example of the code generated is shown in Section 12.6. The measurements presented in Section 12.7 demonstrate that with all the optimizations discussed in this part of the book good code can indeed be generated.

12.1 Representing the ABC machine components

The first step in the realization of the ABC machine on concrete hardware is the design of the data structures that represent the several ABC machine memory components. This involves deciding how to map these data structures onto the concrete memory storages of the target machine. As already stated in the introduction, one of the main differences between the abstract machine and a concrete machine is that the latter has to deal with a limited amount of memory. So the data structures have to be chosen such that memory management of the components can be done efficiently. Furthermore, it is very important to achieve fast access on the elements of the several stores.

The actual design of the data structures and their mapping to the memory will be highly dependent on the architecture of the concrete machine. For some classes of architecture the mapping will be more difficult than others. One has to bear in mind that the ABC machine is a mixture between an idealized graph rewriting machine and a stack-based architecture. Therefore, it will be easier to find a suitable implementation for a Motorola-based architecture than for an IBM PC (poor support for a graph store due to its segmented memory) or transputer-based architecture (poor support for stacks due to limited number of registers and lack of suitable addressing modes). For some classes of architecture maybe a slightly different type of abstract machine would be more convenient. However, it is impossible to design an abstract machine that is equally easy to map to all existing kinds of concrete architectures. Concrete architectures for which the ABC machine is a suitable abstraction have:

- a reasonable amount of memory (a couple of megabytes);
- the possibility of reserving a large proportion of this memory to implement the graph store;
- support for the implementation of stacks (i.e. lots of registers with suitable addressing modes).

A concrete memory generally consists of a finite sequence of consecutive memory cells. Stacks and stores of the ABC machine are implemented on a consecutive partition of such a memory (see Section 12.4 for a possible partitioning). The registers of the concrete machine are generally used to store program counter and stack pointers and to store intermediate arithmetic results. Input and output have to be interfaced with the file system and the I/O facilities offered by the underlying operating system.

The program store

The program store contains concrete machine instructions:

- code produced by the code generator by translating rule-dependent ABC instructions;
- the run-time system including code for memory management like garbage collection.

The several entries (instr-ids) are represented by addresses in this store.

The program counter

The program counter of the ABC machine can of course be represented directly by the program counter of the concrete machine.

The A-stack, B-stack and C-stack

To make the mapping of ABC instructions to concrete instructions convenient it is necessary that a memory partition is allocated for each stack. For each stack a pointer pointing to the top element on the stack is needed. For reasons of efficiency these pointers are usually kept in registers.

Stacks on concrete machines cause the following memory management problems. First of all, concrete stacks have a finite size. So it is necessary to do a run-time *boundary check*. Some machines have hardware support for such a boundary check. Otherwise, additional code has to be generated to do a boundary check when something is pushed on a stack. This will slow down the performance (typically by 20–30 %). Another possibility, taken by many implementations of imperative languages, is simply to leave out the boundary check and take the risk that the system might crash. Other compilers generate boundary checks optionally. When a boundary check is implemented one can either stop the program when the limit is reached or try to dump part of the stack to secondary memory.

Another disadvantage of a stack is that at run-time it may occupy only a very small part of the partition that is reserved for it. In that case the other part of the partition is wasted. With three stacks three parts may be wasted. However, two stacks can be put in one partition in which they grow in opposite directions. In this way they share the free space. In Figure 12.10 this solution is chosen for the A- and B-stacks. Another solution is to allow the programmer, using a trial-and-error method, to specify the amount of memory that should be reserved for each stack. In this way the partitions for a specific application can be tuned by hand such that the stacks are big enough but the amount of wasted memory is minimal.

Another way to reduce the problems mentioned above is by reducing the number of stacks. It is not really necessary to implement every stack separately. It is also possible to merge two or even all three of them into one stack. The merging of the B- and C-stacks causes no problems and is certainly advisable when parallel code has to be generated (see Chapter 17). However, merging the A-stack has to be done in such a way that it is possible to recognize the elements belonging to the A-stack, since they are needed for garbage collection (see Section 12.2). This generally requires additional space for tagging stack elements and may cause a slightly more complex access or a more complex garbage collector. If one decides to merge stacks the code generation will get slightly more difficult, since the offsets of the ABC push and pop instructions will not correspond directly to offsets on the concrete stacks. To make code generation simple again one can incorporate the merging of stacks at the ABC level. One then obtains an A-B&C machine or an A&B&C machine. The translation from Clean to such a changed abstract machine with merged stacks is no more difficult than the code generation for the current ABC machine.

The graph store

The graph store is represented in a memory partition called the **heap**. A graph is composed of nodes. Node-ids are just addresses of memory cells. In principle there are many different choices possible for the representation of ABC nodes in memory. However, the ABC code that is generated by a Clean program (see Chapter 11) assumes that the node on the bottom of a stack frame can always be overwritten with another node. This can be implemented simply in a conventional memory when all nodes are equal in size. The problem is that Clean nodes have variable arity. To solve this problem without wasting memory one can split up an abstract node into two separate concrete nodes: a fixed-sized part and, optionally, a variable-sized part (see Figure 12.2).

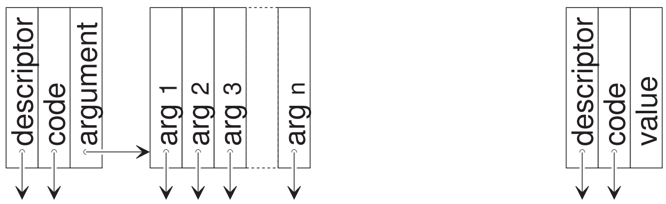

Figure 12.2 Two possible node structures.

The fixed-sized part generally contains:

- an entry in the descriptor store, the **descriptor pointer**, uniquely representing the corresponding Clean symbol (see below);
- a code field containing an address in the program store, called the **code pointer** (commonly pointing to the corresponding node entry);
- a field containing either the address of the variable-sized part or a basic value. The address is called the **argument pointer**.

The variable-sized part generally contains pointers. Each pointer refers to a node representing an argument of the symbol. It can also contain a basic value (string). In principle, no additional tags are needed in nodes. The split representation of nodes makes it always possible to overwrite the fixed-sized part with another fixed-sized part. This representation is essentially equivalent to (and developed independently from) the one chosen by Peyton Jones and Salkild in their spineless tagless G-machine paper (Peyton Jones and Salkild, 1989).

As is the case with stacks, the graph store in the ABC machine is unlimited. In a concrete machine that is not the case. When a new node has to be created in a full concrete graph store one can try to reuse parts of the heap filled with nodes that have become garbage because they are not connected to the data graph any more (see Chapter 5). This well-known technique is called **garbage collection**. A piece of code that takes care of this is called a **garbage collector**. Solutions for garbage collection are discussed in Section 12.2.

The descriptor store

Just as in the ABC machine an actual descriptor in the descriptor store should contain:

- the symbol name (as a string);
- the arity of the symbol;
- the address of the apply entry in the program store.

Furthermore, it is convenient for memory management reasons to store additional information on the kind of node, such as whether it contains a basic value and if so, of what type. The type and arity of a symbol can be used by a garbage collector to determine the structure of a node. The apply entry is used for curried functions.

The input/output channel

As soon as the reduct of the start rule has reached a root normal form the root symbol is written to the standard output channel of the concrete machine.

12.2 Garbage collection

In this section the implementation of garbage collection is discussed in the context of the realization of the ABC machine. In the literature many kinds of garbage collection algorithms can be found. It depends on the available hardware, on the demands of the user and on the properties of the programs which of these algorithms is suited best. Below several well-known algorithms are briefly discussed and their suitability for the collection of variable-sized Clean nodes stored in the heap is examined. First some terminology is introduced.

Terminology

In this section on garbage collection a **node** is defined as a number of contiguous computer cells that can be manipulated by the user program. For a garbage collector the representation of a Clean node as given in the previous section can actually consist of two nodes in memory: the fixed-sized part and the variable-sized part. If a node x contains a pointer to a node *y*, then *x* is called the **parent** of *y*, and *y* is called a **child** of *x*. Nodes that cannot be reached any more via pointers from the root(s) of the graph are called **garbage**.

The *storage allocator* and *garbage collector* together manage the memory. The **storage allocator** is a program that assigns storage (nodes) to the user program. For the storage allocator the memory consists of two parts: a part with unallocated nodes, the **free space**, and a part with allocated nodes. The storage allocator takes storage from the free space and makes it available to the user program in the form of nodes. The **garbage collector** retrieves allocated nodes that are not being used any more and reassigns them to the free space.

Garbage collection of nodes that are all of the same fixed size is generally easier than garbage collection of nodes that vary in size. In the proposed realization of the ABC machine the nodes in memory can have different sizes (due to the variable-sized parts). Therefore only garbage collectors are considered that can deal with variable-sized nodes.

Management of the free space

The free space generally consists either of a *linked list of unused nodes* or it consists of a *contiguous part of memory*.

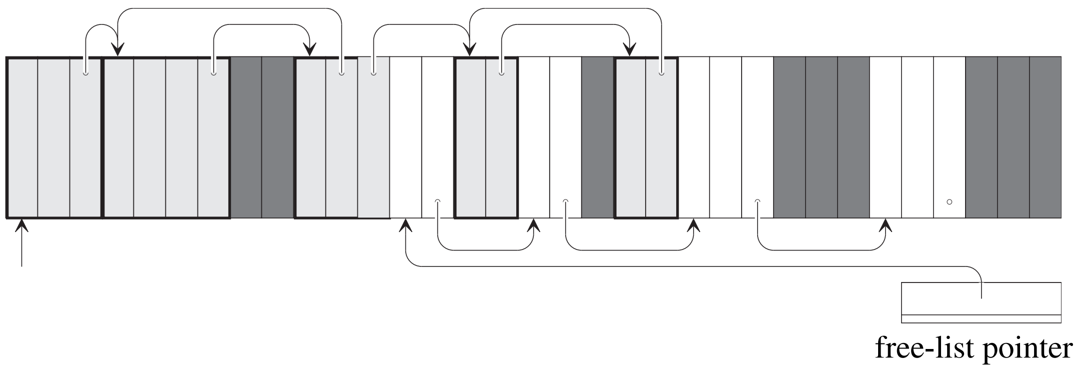

Figure 12.3 Free space management with a free-list. White cells are unused, light-grey cells are nodes in use, dark-grey cells are garbage.

When the free space is a *linked list*, a pointer to the head of the free-list is maintained, for instance in a register (see Figure 12.3). Such a linked list is called a **free-list**. The main disadvantage of a free-list is that storage allocation is more complex when one has to deal with variable-sized nodes. In general one has to walk through the linked list to find an unused node of a certain size. It can even happen that there are enough nodes available in the free-list, but none of them has the required size. So although there is space enough in the heap it cannot be used because of this **memory fragmentation.**

When there is no piece of contiguous memory of the right size available in the free-list one can move the available nodes in the memory such that they form a contiguous storage area. This process is called **compaction**. Compaction is a costly process since all nodes must be moved and pointers have to be adjusted such that the free-list fills a contiguous piece of memory. So it is important to avoid or postpone compaction as much as possible.

Morris (1978) introduces a compaction algorithm for variable-sized nodes. It does the compaction in linear time with respect to the number of non-garbage nodes and needs only one additional bit per pointer. The algorithm requires two scans. The first one only readjusts forward-pointing references. The second one updates references pointing backwards and does the compaction.

One way to postpone compaction is by keeping multiple free-lists (instead of one): i.e. besides the normal free-list with nodes of miscellaneous sizes a separate free-list is maintained for each node size commonly used in a program. An unused node is returned to the appropriate free-list. Requests for new nodes are handled according to their size. If the only node available is larger than needed, it is split into two nodes; the first is used to satisfy the request, the second is returned to one of the free-lists.

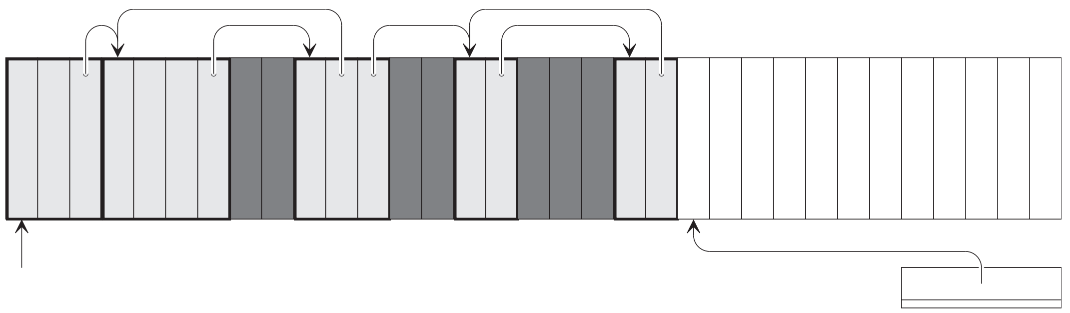

Figure 12.4 The free space contained in a contiguous part of memory.

When the free space is always a *contiguous part of memory*, the storage allocator is very simple: a pointer to the next free word is maintained (see Figure 12.4). If a request for a node with a size of n words is done, this pointer is simply moved n words further on. So having a free space that consists of a contiguous part of memory is ideal for storage allocation for variable-sized nodes. Another advantage of using a contiguous part of memory is that no compaction is needed.

The garbage collector

A garbage collector has the following tasks: the identification of the nodes that are garbage and the collection of these nodes into the free space. If the free space consists of a free-list it may be necessary to compact collected nodes such that they form a contiguous storage area (after or during the incorporation of the nodes into the free space).

One of the problems of garbage collection is that the garbage collector is invoked at the moment that there is no space available any more, perhaps not even for the garbage collector itself. So all the space that the garbage collector needs must be reserved in advance. A practical algorithm runs in a fixed and reasonable amount of space. Garbage collectors can be divided in two groups: *stop-and-collect* garbage collectors and *continuous* garbage collectors.

Mark–scan and *copying* garbage collectors belong to the stop-and-collect category. With **stop-and-collect** garbage collectors the execution of the regular user program is suspended when the user program asks for a new node and the free space contains not enough words to satisfy the request. The garbage collector is called to retrieve unused memory in the free space. Systems with a large (virtual) memory spend a fair amount of their time in garbage collection. The garbage collection time can be up to 30% of the CPU time.

For interactive systems stop-and-collect collectors may lead to intolerably long interruptions. This drawback can be attenuated by using **continuous** garbage collectors that continuously update the free space. Example of continuous collectors are *reference count* garbage collectors or *on-the-fly* garbage collectors.

With a reference count garbage collector the number of pointers to a node is administered in the node and updated when this number changes. In this way the collection is naturally distributed over time. A disadvantage of reference count garbage collectors is that they cannot reclaim cyclic graph structures.

On-the-fly garbage collectors are separate *processes* that can run interleaved or in parallel with the user program, thus distributing the pause caused by the garbage collector over time. Both *on-the-fly copying* as well as *on-the-fly mark–scan garbage* collection are possible.

12.2.1 Copying garbage collection

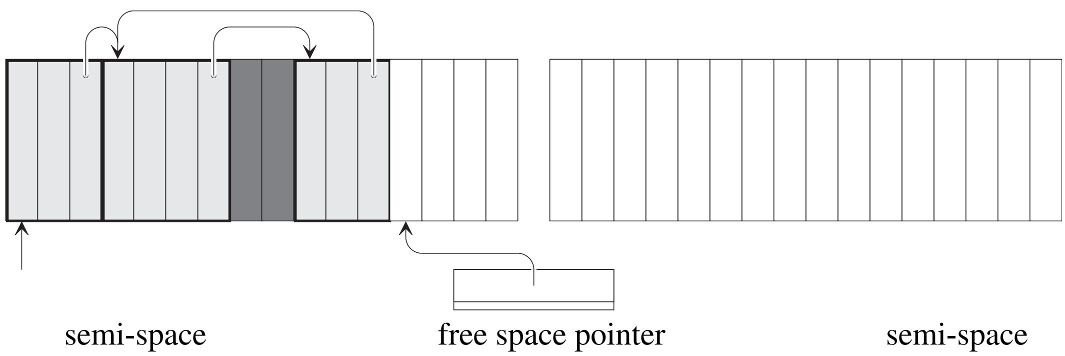

Figure 12.5 Semi-space organization for a copying garbage collector.

Copying garbage collectors generally split the heap into two areas called **semi-spaces**. At any given time, the storage allocator allocates nodes in only one of these semi-spaces. The free space consists of a contiguous part of memory in the active semi-space (see Figure 12.5).

When this free space gets exhausted, all non-garbage nodes are copied from one semi-space to the other unused one. From then on the allocator allocates nodes in the other semi-space until this gets exhausted, and so on. For the actual copying of the nodes there are many different algorithms that use or do not use a stack and/or markings in the nodes. Here only one copy algorithm is treated, which uses neither a stack nor marking bits. A more detailed description of the algorithm can be found in Minsky (1963).

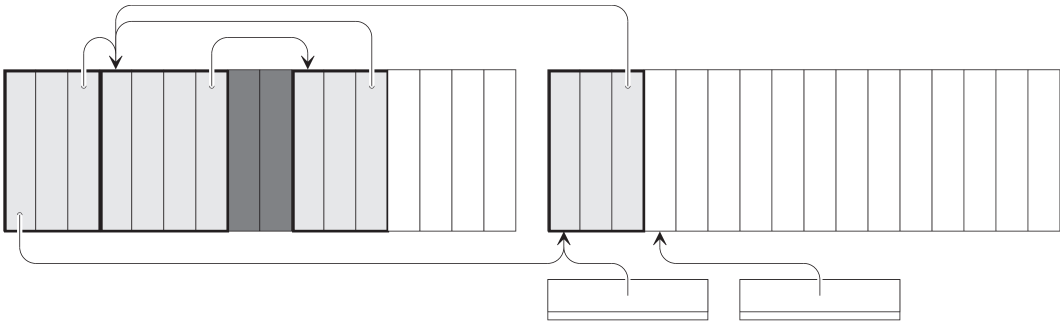

Figure 12.6 Snapshot: copying the used nodes in the unused semi-space.

The algorithm proceeds as follows when the old semi-space gets exhausted:

- First, all roots of the non-garbage graphs are copied to the new semi-space (Figure 12.6).
- Then, two pointers are positioned in the new semi-space, one at the beginning and one at the end of the roots. The pointers are used as follows: the first one, say *F*(irst), indicates the node of which the children are next to be copied to the new semi-space; the last one, say *L*(ast), indicates where copied nodes are to be put.
- All children of the node at *F* are now copied to the new semi-space *L*, and *F* is moved to the next node. This is repeated for the children of the next node, and so on. The copying algorithm terminates when *F* becomes equal to *L*. In this copying process, special precautions have to be taken to maintain sharing of nodes, and to guarantee termination if there are cycles in the graphs. Therefore, a node that resided in the old semi-space but which has been copied to the new semi-space, is overwritten with the address of its copy in the new semi-space. This address is called a **forwarding address** or **forwarding pointer**. Before a node is copied to the new semi-space, it is checked whether it contains a forwarding address.

If not, it is simply copied; if so, the pointer in its parent is changed to the new address.

In general, with a copying garbage collector twice as much storage is needed as the allocator has at its disposal. This is a very large amount, but it is fixed, so it can be reserved in advance. Furthermore, there is no other space needed nor any other stack allocation nor extra bits nor a free-list administration nor compaction, since in the new semi-space the non-garbage nodes are in a contiguous part of the memory. A copying garbage collector is very fast when there is a lot of garbage in the semi-space. But the performance drastically decreases if there is only a little garbage: the number of copy actions depends on the number of used nodes.

In the basic copying garbage collection scheme all accessible nodes are copied to the new semi-space when the old one gets exhausted. This implies that nodes that live very long are copied frequently from one semi-space to another. When such objects occupy much memory it is useful to optimize a copying garbage collector by avoiding the copying of older objects. However, generation scavenging (Liebermann and Hewitt, 1983), which is the best-known optimization of this kind, cannot be used since it is only of great use when younger nodes generally point to older nodes. Unfortunately, in graph rewriting systems older nodes generally point to younger ones.

12.2.2 Mark–scan garbage collection

A mark–scan garbage collector does not split up the heap, but uses a free-list. Hence, when variable-sized nodes are collected, fragmentation can occur (see above) and compaction is needed occasionally.

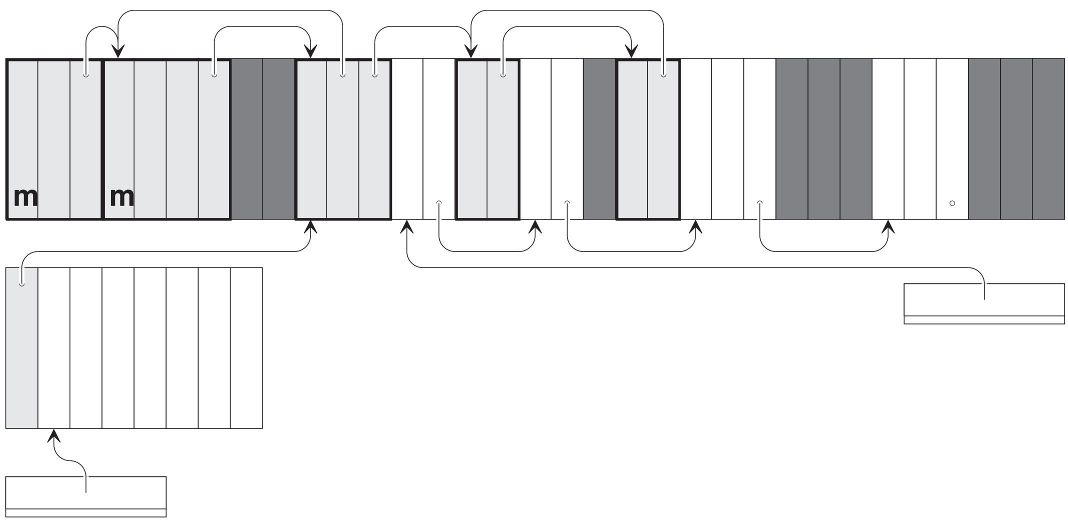

Figure 12.7 Mark–scan garbage collection: phase 1 using a recursive graph traversal stack.

The collector does its job in two phases. In the first phase, the **mark phase**, the accessible nodes are marked. Starting from all roots, the complete subgraphs reachable from these roots are traversed recursively and all encountered nodes are marked (see Figure 12.7). The reachable nodes can be marked by the following algorithm:

```
push all roots on a stack;
WHILE the stack is not empty
DO    mark the node indicated on top of the stack;
      replace it by all unmarked children
OD
```

In the second phase, the **scan phase**, the whole storage is scanned, node by node. Marked nodes are unmarked again, and unmarked nodes, which are garbage, are added to the free-list (see Figure 12.8).

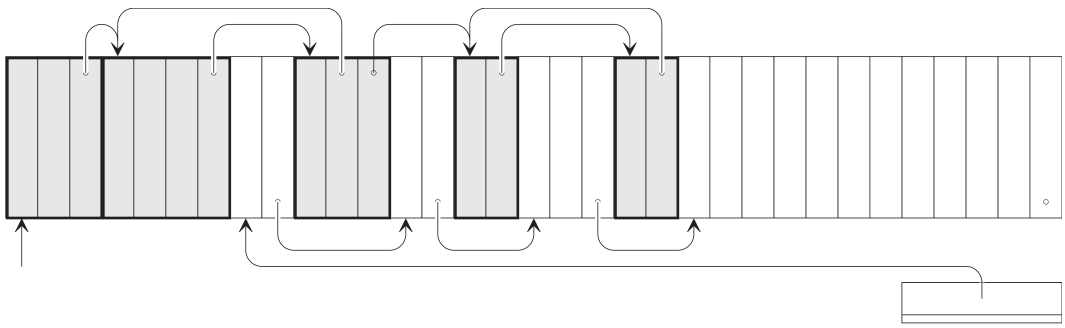

Figure 12.8 Mark–scan garbage collection: phase 2.

Mark–scan garbage collection is not as efficient as copying. In the worst case the complete heap has to be traversed twice: once for the mark and once for the collect phase. When the heap contains little garbage the performance decreases considerably. The mark phase has much more work to do and a large amount of additional space is needed for the recursion stack. In the worst case the size of the extra recursive graph traversal stack equals the total number of nodes available in the memory when there is no garbage and all nodes form a single linked list. However, there are several optimizations that use only a small, fixed-size stack or even no stack at all (by using pointer reversal). These algorithms and many others are discussed in Cohen (1981).

12.2.3 Reference count garbage collection

When garbage collection is done with reference counts, an additional field, a **reference count**, is reserved in each node. In this field the number of pointers to the node is administered. The reference count field must be large enough to hold the total number of nodes in the heap.

When nodes are generally small, this is a significant space overhead. Reference count garbage collectors use a free-list. So compaction may be needed once in a while.

All node creations and pointer manipulations are done by special procedures that not only perform the creation of the node and the pointer manipulation itself, but also maintain the correct reference count:

- new nodes are created with reference count 1;
- when a pointer to a node is copied, the reference count of the node is incremented by 1; when a pointer to a node is deleted, the reference count of the node is decremented by 1;
- when the reference count becomes zero, the node becomes garbage and can be added to the free-list. The reference count of all the children of the node can now also be decremented and they may become garbage too if they are not shared. A recursion stack is needed to adjust all children of a deleted node.

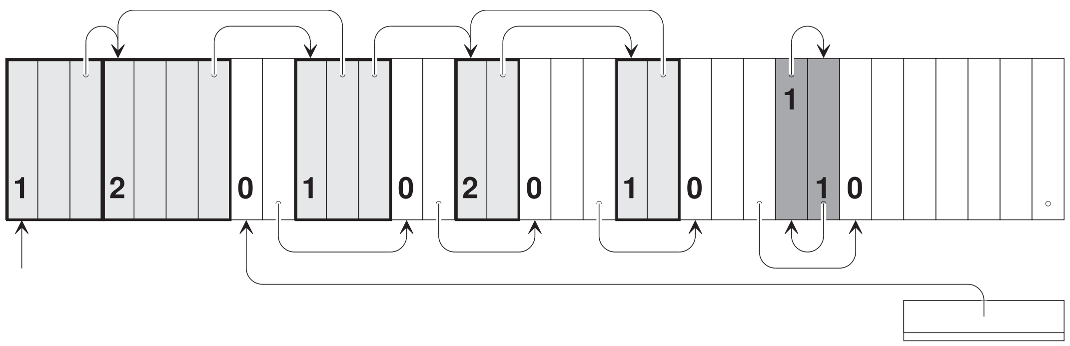

Figure 12.9 Reference count garbage collection. The dark grey cyclic structure cannot be collected.

The advantage of reference count garbage collection is that garbage can be collected immediately after it is created. The cost of reference counting garbage is independent of the proportion of the heap that is used. However, there are some severe drawbacks. For small nodes there is a significant space overhead in the form of the reference count. With each pointer manipulation an update of the counter has to be done. This means that, even if no garbage collection is needed, administrative work must still be done. Another important disadvantage of reference counting is that it is impossible to recover cyclic structures. In a cyclic structure all nodes have reference count greater than zero (see Figure 12.9). When such a cyclic structure becomes garbage it cannot be detected unless the whole memory is examined again.

An optimization that avoids the deletion recursion stack is called lazy garbage collection (Glaser and Thompson, 1985): when a node becomes garbage, it is added to the free-list. When the reference count field is large enough to hold a pointer, it can be used for the linking. Now, when a node is needed, it is taken from the free-list and its reference count is set to one. Next, the reference counts of the children of the node are decremented. The children that become garbage are put in the free-list in the same manner. In this way lazy garbage collection is obtained with the major advantage that the collection is only done when needed, while there is no need for a recursion stack.

A solution for the collection of cyclic structures is to combine the reference count technique with stop-and-collect garbage collection. The former would be used during most of the processing time. The latter would be used as a last resort when the storage area is fouled up with non-recyclable nodes. When reference count is combined with other methods, the reference count field does not have to be of maximum size. A smaller size can be used. When the smaller reference count field overflows, the nodes cannot be reclaimed by the reference count algorithm. Just like cycles, such nodes are reclaimed by the other method.

12.2.4 On-the-fly garbage collection

Another possible solution for continuous garbage collection is to have a separate garbage collection process (an **on-the-fly** garbage collector) that works concurrently (i.e. interleaved or in parallel) with the user program. Of course, for parallel garbage collection a second dedicated processor is required. The garbage collection is divided into two concurrent processes: the user program, in this context usually called the **mutator**, and the **collector**, which performs the marking/copying and the collection of the garbage. Being concurrent processes, some communication between mutator and collector is necessary.

The best-known on-the-fly garbage collection method, introduced by Dijkstra *et al.* (1978), is based on the *mark–scan* method. The mutator colours nodes to be examined by the collector. Therefore, three mark colours, white, grey and black, are introduced. White corresponds to no marking, and black to the marking done by the collector. The mutator only marks nodes grey to indicate to the collector that marking has to be done from this node on.

Copying garbage collection, which incorporates compaction via semi-spaces, can also be done on-the-fly (Baker, 1978). In this case the mutator must know that there are two semi-spaces in which nodes can reside. Allocation of new nodes is done in the semi-space to which the collector currently is copying all nodes.

On-the-fly garbage collectors are best suited for interactive programs. But, as with reference count garbage collection, they are also inefficient when they are executed in an interleaved manner. This is due to the communication and synchronization overhead between the mutator and the collector.

12.2.5 Comparison of the methods

When there is enough memory available and there are no strong requirements on the real-time behaviour of the program, copying garbage collection is generally the best method because it is a very simple and fast method (see Table 12.1).

Table 12.1 Summary of the different kinds of garbage collection methods: '+' means good, '–' means not so good, 'o' means 'neutral', '+/–' means good when there is a large amount of garbage and '–/+' means good when there is little garbage.

	Stop-and-collect		*Continuous collectors*		
	Copying	*Mark–scan*	*Reference count*	*On-the-fly copying*	*On-the-fly mark–scan*
Time efficiency	+/–	–/+	o	+/–	–/+
Space efficiency	–	+	–	–	+
Free-list not needed	yes	no	no	yes	no
No time overhead if space free	yes	yes	no	no	no
Collects cycles	yes	yes	no	yes	yes

If the amount of memory is too limited, mark–scan can be used instead of copying. A good solution is the following: implement both a copying as well as a mark–scan collector and automatically switch between them, depending on the actual memory consumption. If the real-time behaviour is very important one can use an on-the-fly variant of copying or mark–scan. A pure reference count collector is less desirable, because it cannot deal with cyclic structures.

When garbage collection has to be performed in a parallel environment, other problems can occur and the comparison between the algorithms will be different (see Chapter 17).

12.3 The Motorola MC68020 processor

There are two reasons for choosing the Motorola 680x0 family of processors (68010, 68020, 68030, 68040 …) as target machine (Motorola, 1984–1985). First of all, they have been used in several widespread machines such as the Sun3, Apple Macintosh and Atari ST. Furthermore, the Motorola processors are very suitable as an actual target machine to illustrate how the ABC machine (and therefore functional languages in general) can be implemented efficiently. It should be no problem to use the ideas presented when generating actual target code for other register based processors (such as the Intel 80x86 family).

In this chapter the instruction set of the Motorola MC68020 processor is used as example. The MC68020 processor is a 32–bit machine which makes a large heap possible. Besides a *program counter* and a *status register* it contains two kinds of general purpose registers, namely *data registers* and *address registers*, eight of each kind. The status register contains the following condition codes: N (negative), Z (zero), V (overflow) and C (carry). The data registers, often denoted by d0–d7, are mainly used in arithmetical operations whilst the address registers (indicated by a0–a7) can be used to access data structures that are kept in memory. The address registers can be used in combination with many different addressing modes such as post-increment and pre-decrement. So stacks can be implemented very efficiently.

Summary of the addressing modes used in the examples:

(address)	take the contents of this address;
offset(address)	take the contents at address + offset;
(address)+	take the contents of this address and then increment the address in the register;
–(address)	first decrement the address in the register and then take the contents of this new address.

For the remaining parts of this chapter no further knowledge is assumed about the specific architecture of the MC68020 processor. Some general familiarity with assembly language will suffice to understand the examples of concrete code given in this section.

The following MC68020 machine instructions are used in the examples:

bra	branch; same as jmp in ABC machine, address must be nearby;
bcs	branch when the carry bit (C) is set in status register;
bmi	branch when negative bit (N) is set in status register;
beq	branch when equal bit (Z) is set in status register;
bne	branch when equal bit (Z) is not set in status register;
bsr	branch to subroutine; same as jsr, but address must be nearby;
jsr	equivalent to jsr in ABC machine;
rts	equivalent to rtn in ABC machine;
add.l	integer addition (32 bits);
addq.l	addition of small integer values (8 bits);
sub.l	integer subtraction;
subq.l	subtraction of small integer values;
muls.l	integer multiplication;

move.l	move a 32 bit long word from the source to the destination;
lea	load a 32 bit memory address in a register;
cmp.l	compare two 32 bit long words;
seq	set all bits in byte operand when Z is set; otherwise reset all bits;
extb.l	extend sign of byte to long word;
tst	test, compare with zero.

12.4 Representing ABC components on the MC68020

In this section we present a mapping of the basic ABC machine components onto a specific machine, the Motorola MC68020. The mapping of the different ABC memory storages onto the concrete memory is given in Figure 12.10.

The representation of the components of the ABC machine (i.e. ABC stacks, graph store (heap) and descriptors) on the MC68020 does not cause many difficulties. Stacks can be implemented simply using some of the address registers. Implementing the heap is somewhat more complicated. The structure of the descriptors mainly depends on how higher order functions are implemented.

The program store and program counter

The mapping of the program store and program counter is trivial: their concrete counterparts on the MC68020 can be used.

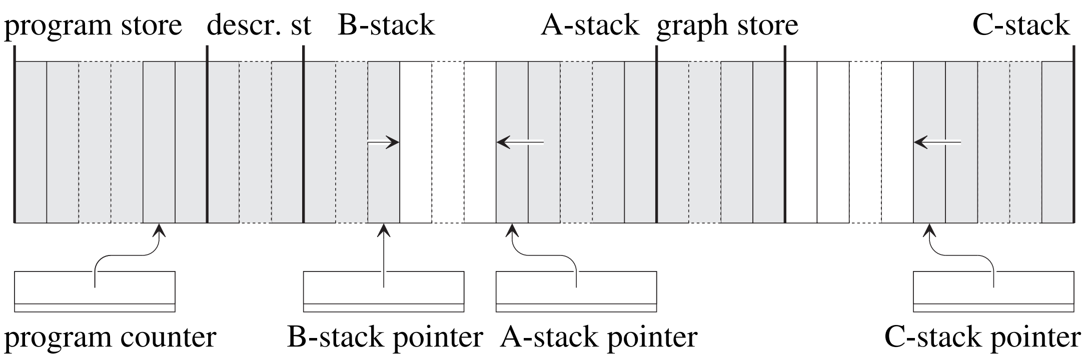

Figure 12.10 Mapping the ABC memory stores on a concrete linear store.

The A-stack, B-stack and C-stack

For the C-stack the system stack of the MC68020 is taken (i.e. the stack used by the processor itself when it performs a subroutine call). So the jmp and rtn instructions of the ABC machine can be mapped directly to those of the MC68020 (of course, for the jsr_eval instructions more things have to be done; see below). This implies that address register a7 (normally called sp (stack pointer)) is reserved as usual. The A- and B-stacks are allocated in one contiguous area of memory in such a way that they can grow in opposite directions. In this way a check on stack overflow of both the A- and B-stacks can be done with a few instructions (just compare the two stack pointers and check that their difference is not negative). The pointers to the tops of the stack are held in registers: for the A-stack register a3 is reserved, for the B-stack register a4 (from now on these registers will be called asp and bsp).

The graph store

For the graph store a *contiguous* area of memory is chosen and not a free-list. The pointer to the free area is stored in register a6 (called hp (heap)), whereas the number of free long words (i.e. 1 long word = 4 bytes) is stored in register d7 (fh from now on (free heap)). As explained in Section 12.2, with this representation the allocation of memory becomes cheap. Filling newly created nodes can be done very efficiently.

Suppose that a Cons node should be created that has two arguments. The references to both arguments are kept in the address registers a1 and a2. At the end a pointer to the new node is returned in register a0. First we have to check whether there is enough space in the heap. This is done by:

```
subq.l  #4, fh    ;  4 long words needed to store the new node
bcs     call_gc   ;  call the garbage collector if not enough free space
```

Now the heap pointer (held in register hp) refers to the first free long word in the heap. Filling the node by using hp is straightforward (for the actual representation of nodes see below):

```
return_from_gc:
move.l  hp, d0           ;  first the variable part is filled
                         ;  a pointer to it is temporarily stored in d0
move.l  a1, (hp)+        ;  store pointer to the first arg. in variable part
move.l  a2, (hp)+        ;  store pointer to the second arg. in variable part
                         ;  now the fixed part is treated
move.l  hp, a0           ;  store a pointer to it in a0
move.l  #Cons, (hp)+     ;  the descriptor field must point to entry of Cons
move.l  d0, (hp)+        ;  store argument pointer
```

Representation of nodes

As described earlier, a node in the ABC machine consists of fixed- and variable-sized parts. A drawback of the ABC node structure as suggested in Section 12.1 is that the size of the nodes is relatively large: the fixed part consists of 3 long words (12 bytes), one for the pointer to a descriptor, one for the code pointer and one for the pointer to the variable-sized part. It is important that nodes are as small as possible: creating, filling and copying of nodes can be done faster and because less memory is consumed the garbage collector will be called less often.

One can observe that if a node is in root normal form, its code pointer always points to the root normal form code. In that case only the pointer to the descriptor is of interest. On the other hand, if a node contains an (unevaluated) expression, only the code pointer is essential: the descriptor is not used. This observation makes it possible to combine the descriptor and code pointer into one field. This reduces the size of the fixed part of a node by one third. There remains one problem: the arity of a node, stored in the descriptor table, is needed for garbage collection. This information is not available when a code pointer is stored in the node. The problem can be solved by storing the arity in the program store, just before the node entry.

The space-saving new node structure is illustrated in Figure 12.11. The disadvantage is that a tag has to be inserted: the highest bit of the first word of the code/descriptor field (note that this field consists of two words) indicates whether it contains a descriptor or a code address. If this bit is set, the second word is an index in the descriptor table. Otherwise, the code/descriptor field contains a code pointer that is used to reduce the node to root normal form.

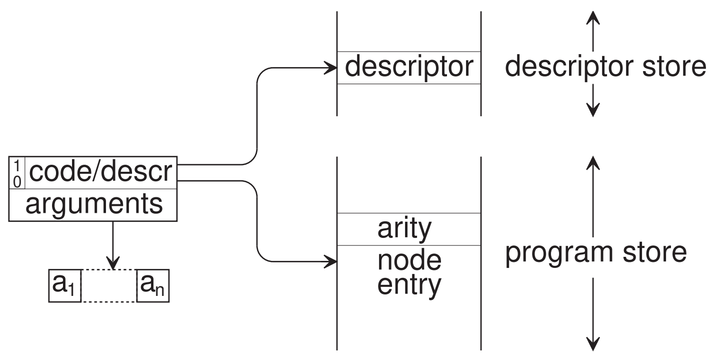

Figure 12.11 The structure of nodes and descriptors

In the ABC machine nodes can be evaluated to root normal form by means of the jsr_eval instruction. This instruction fetches the code address from the node on top of the A-stack and jumps to it. Owing to the combined code/descriptor we first have to check whether the node is already in root normal form before the jump to the evaluation code is made. The next piece of code shows how this can be achieved (assume that register a1 refers to the node to be reduced).

```
move.l  (a1), d6    ;  get the code/descriptor field
bmi     is_in_rnf   ;  check whether the highest bit is set
save all the registers in use
move.l  a1, a0      ;  ensure that a pointer to the node that
                    ;  is going to be evaluated is in reg. a0
move.l  d6, a1      ;  move the evaluation address in a1
```

```
jsr      (a1)      ; call the evaluation code
move.l   a0, a1    ; move the result of the evaluation back in a1
restore the previously saved registers
is_in_rnf:  …
```

The loss of efficiency caused by the introduction of the tag is very limited while a lot of space (and hence time) is saved with this alternative representation.

When a node is not in root normal form, this alternative representation leads to slightly less efficient code (an extra move instruction and a conditional branch are needed). But when the node is already in root normal form the code becomes much faster. In that case the saving and restoring of the registers are not needed any more.

Garbage collection

Memory is recycled by a *copying garbage collector* that uses two semi-spaces (see Section 12.2). When one semi-space is filled up the garbage collector copies all the nodes still needed for the execution of the program to the other semi-space, leaving all the garbage behind.

The descriptor store

In Section 11.2 the ABC code for the apply function is given (_AP). The code shows that at run-time two arities are needed (the number of arguments collected in the node so far and the number of arguments needed). The apply entry of the curried function has to be called when all arguments are supplied. All this information has to be found via the descriptor that points to an entry of the descriptor table (Figure 12.12).

A straightforward translation of the ABC apply code will result in rather inefficient MC68020 code. To increase the efficiency a pointer is stored in the descriptor table that points to the code to be executed when the curried function is applied to an additional argument.

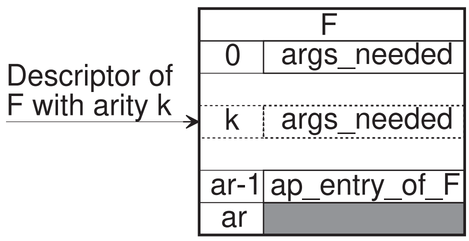

Figure 12.12 Lay-out of a descriptor in the descriptor table.

When F is a symbol defined with arity ar its descriptor contains a string representation of the name and ar+1 entries. The pointer stored in the descriptor field of a node is one of these entries. It indicates with which arity F is actually applied, which is used by the garbage collector. Furthermore, the entry contains a reference to the code to be reduced when a partial application of F is applied to an additional argument (with the (internal) apply function).

With the aid of the previous representation the translation of the apply code will result in the following ABC instructions (it is assumed that register a1 refers to the node containing the partial application and register a5 refers to the beginning of the descriptor table):

```
move     2(a1), a2   ;  get the offset of the descriptor entry
add.l    a5 a2       ;  add this offset to the beginning of the descriptor table
move.l   2(a2), a2   ;  retrieve the reduction code
jsr      (a2)        ;  call the reduction code
```

12.5 Generating MC68020 code

A straightforward way of generating concrete machine code is by means of *macro expansion*: each ABC instruction is considered to be a macro application that is substituted by a sequence of MC68020 instructions. The main disadvantage of this method is that the context in which the instruction appears is not taken into account.

An illustrating example is given by the next piece of ABC code together with macro expanded MC68020 code. Assume the following macro definitions:

```
#macro    push_b(n)       move.l   –((n+1)*4)(bsp), (bsp)+
#endmacro
#macro    addI            move.l   –(bsp), d0
                          add.l    d0, –4(bsp)
#endmacro
```

and the following ABC instructions:

```
push_b  0
push_b  2
addI
```

Applying the macro definitions will result in:

```
move.l   –4(bsp), (bsp)+
move.l   –12(bsp), (bsp)+
move.l   –(bsp), d0
add.l    d0, –4(bsp)
```

However, if the three ABC instructions were considered simultaneously, one could use the fact that the MC68020 add instruction does not require that the arguments are on top of the B-stack. By looking at the context in which instructions appear, the movement of data can be reduced. A more efficient code generator might compile the three ABC instructions into the following three MC68020 instructions, which are about 30% faster than the ones above:

```
move.l  –4(bsp), d0
add.l   –8(bsp), d0
move.l  d0, (bsp)+
```

So before generating code it is useful to group ABC instructions into blocks. These **basic blocks** can be considered as kinds of atomic action. They specify state transitions that convert the state of the ABC machine (which is determined by the contents of the stacks and the graph store) at the beginning of these blocks into the final state at the end of the basic blocks. Now the task of a code generator becomes to implement such actions as efficiently as possible.

The largest gain will be achieved when these basic blocks are as large as possible. In the current Clean code generator a basic block consists of the maximal sequence of ABC instructions that does not contain any label definitions or instructions that might change the flow of control (e.g. subroutine calls or conditional branches). Basic blocks can be made larger by replacing a subroutine call by its code. However, programs become larger in this way, and therefore code substitution is only advisable when the substituted code is relatively small.

12.5.1 Compile-time optimizations

With the aid of basic blocks the compile-time analysis of ABC programs is simplified. A few examples follow.

Flow of control

The original ABC instructions specify an evaluation order. Grouping these instructions into basic blocks allows us to deviate from this order as long as the new evaluation has no effect on the final result obtained at the end of the basic block (i.e. the new code sequence should specify the same state transition). Changing the evaluation order makes it possible to improve the generated code.

Suppose that d0+d1 and d0+d2 have to be computed in a register while d1 is used after these computations, but d0 and d2 not. First computing d0+d1 and then d0+d2 will give:

```
move.l    d0,  d3
add.l     d1,  d3
add.l     d2,  d0
```

It is better to compute d0+d2 first. It saves one instruction and one register.

```
add.l     d0,  d2
add.l     d0,  d1
```

In the ideal case the code generator will determine an evaluation order of instructions such that the execution costs of the generated code are as low as possible. The execution costs can be found by simply adding the execution times of all the individual instructions (one should note that the execution time of an instruction may depend on the preceding instructions). But the problem of finding an evaluation order in such a way that the total time is minimal is NP-complete, which makes an algorithm based on this strategy useless.

Optimize the use of the registers

A different approach is not to minimize the execution time but to minimize the number of registers needed to evaluate a basic block. This is a reasonable approach because, since the registers of a processor are relatively sparse, the quality of the generated code will strongly depend on how well they are utilized. Table 12.2 summarizes which of the 8+8 registers of the MC68020 have been reserved for special purposes.

Table 12.2 Summary of reserved registers

Mnemonic	*Register*	*Function*
fh	d7	number of free words in heap
asp	a3	pointer to the top of the A-stack
bsp	a4	pointer to the top of the B-stack
hp	a6	pointer to the first free node in the heap
sp	a7	pointer to the top of the C-stack

Parameters and results of functions are in principle passed via the A- and B-stacks. However, the efficiency can be increased a lot when registers are used instead. So arguments and results are kept in *registers* whenever possible. Only when there are not enough free registers are the values stored on the stacks.

In the ABC machine the conditional jump instructions base their decision whether or not to jump on the Boolean value that is on top of the B-stack. If this boolean is the result of some comparison (which is indeed often the case) then it is not necessary to calculate this value explicitly. Instead of that, the conditional jump can use the *condition codes* in the status register of the MC68020 that are set implicitly after performing the comparison.

Take for example the following ABC instructions:

```
eqI_b +10 0
jmp_false label
```

The following code would have been generated if the boolean is calculated explicitly (assuming that the element on top of the B-stack is kept in register d0, register d1 is used to calculate the boolean and the value –1 and 0 represent respectively TRUE and FALSE):

```
cmpi.l   #10, d0
seq      d1
extb.l   d1
tst.l    d1
beq      label
```

Much better code is generated using the condition codes of the MC68020:

```
cmpi.l   #10, d0
bne      label
```

Heap allocation

A basic block may contain instructions that reserve space in the heap. Such an instruction has to check whether the heap contains enough free cells. If this is not the case the garbage collector has to be called. Since it is known at compile-time how many cells are actually needed in a certain basic block, all these checks can be done at once instead of performing separate checks for each instruction.

Stack use optimization

The generation of additional instructions for boundary checks of the stacks inside a basic block can be combined. For each block we can determine the maximal amount by which the stack will grow and perform the tests in one place. The same holds for the adjustment of the stack pointer (due to the various push and pop instructions).

Using alternative instructions

On the MC68020 there are several instructions that, when applied to certain arguments, can be replaced by other instructions having the same effect, but with the advantage that their execution time is less. This replacement can easily be performed during the last phase of the MC68020 code generation or even by the assembler.

12.5.2 The code generation process

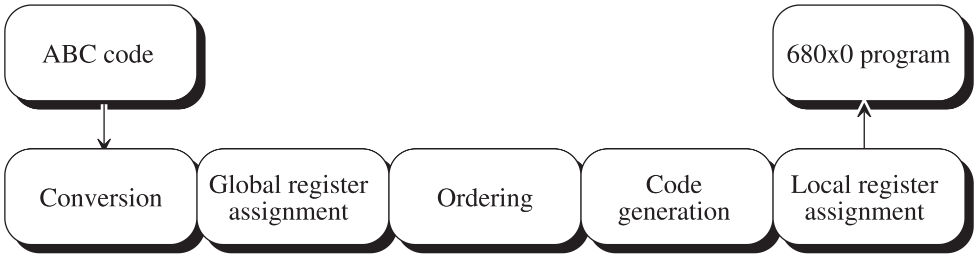

Figure 12.13 The code generation phases.

The actual code generation is divided into a number of phases (Figure 12.13). During the *conversion phase* the initial ABC program is divided into *basic blocks* and converted into an internal representation using graphs. The *global register assignment phase* determines which entries of the A- and B-stacks are kept in registers at the beginning and at the end of these basic blocks. The *ordering phase* determines the order in which all the subexpressions have to be evaluated. During the *code generation phase* (pseudo-) MC68020 code is generated according to the order specified by the previous phase. The only difference between real MC68020 code and the generated code is that in the latter an unlimited number of (virtual) registers are assumed. Finally, the *local register assignment phase* replaces virtual registers by real MC68020 registers.

The conversion phase

The correspondence between the initial stack frame S_b at the beginning of a basic block and the final stack frame F_b at the end is defined by the instructions of that basic block. A directed acyclic graph is a useful data structure for representing basic blocks such that automatic analysis of these blocks can be done more conveniently. Such a dag gives a picture of how the value computed by each statement in a basic block is used in subsequent statements in the block. The dag representation of a basic block that is used has the following properties:

- The leaves either represent constants or entries of S_b.
- All the other (interior) nodes represent applications of ABC instructions. The arguments of these nodes are the representations of the arguments of the corresponding instructions.

- If a node represents an instruction whose result appears in F_b, this node is labelled with an identification of the corresponding entry in F_b. All the other nodes are not labelled.

The next example illustrates the ABC to MC68020 translation process. Consider the following Clean rewrite rule for F:

```
:: F  !INT  !INT  ->   INT;
   F  a     b     ->   – (* a b) (+ a 3);
```

This rule is compiled into the following ABC instructions (only the code for the rule alternative entry is given):

```
F1:
pushI +3
push_b 1          || push a on top of the stack
addI              || add b and 3
push_b 2          || push b on top of the stack
push_b   2        || push a on top of the stack
mulI              || multiply a and b
update_b 1 3      || update the B-stack
update_b 0 2
pop_b 2
subI              || subtract the topmost elements
rtn
```

The strict entry forms one basic block. The dag that is constructed is given in Figure 12.14. The meaning of the additional information stored in the dag is explained later.

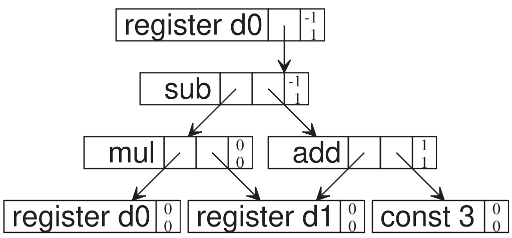

Figure 12.14 The dag of the strict entry.

The global register assignment phase

Global register assignment specifies which values of the initial and final stack frames of each basic block are kept in registers. The information that is used to determine this assignment is obtained from the original Clean program. The Clean compiler uses the type information of the Clean rules to insert special ABC directives in the corresponding ABC program. These directives specify the layouts of both the A- and B-stacks for both the beginning and the end of a basic block.

With the aid of the stack layout information the code generator reserves all the data and address registers that are needed for storing these elements at the beginning of a basic block if such a basic block starts with a label that corresponds to an entry point of a function. The directives describing the layout of the end of a basic block are used to ensure that, when a basic block is left at run-time, the results are kept in the right places (i.e. either in one of the registers or one of the final stack frames itself). The latter may require that the contents of certain registers in use have to be saved on the stack.

The result of the global register assignment of a basic block is administered in the corresponding dag with the aid of two kinds of special nodes, namely *register* and *store register* nodes. A **register node**, which refers to an entry of the initial stack frame S_b, indicates that the value of that entry is kept in a register when entering the basic block (note that register nodes are always leaves of the dag).

> Consider Figure 12.14: owing to the global register assignment, `a` and `b` are kept in, respectively, data registers `d1` and `d0` and the final result should be stored in register `d0`. These register assignments are indicated with the aid of the register nodes.

A **store register node**, which refers to an entry of the final stack frame F_b, indicates that the value of that entry is held in a register when leaving the basic block. So store register nodes are always labelled.

The ordering phase

Under the assumption that none of the instructions in a basic block, except the very last instruction, may produce side-effects, the generation of code can be done independently of the original order of ABC instructions. The only requirement that has to be met is that when generating code for a certain node of the dag all the other nodes that are reachable from this node have already been treated. So we are allowed to change the original evaluation order as long as the previous requirement is fulfilled.

If a basic block does not contain any common subexpression (CSE) (so the corresponding dag is free of sharing), the problem of determining the reduction order such that the number of registers is minimal is simple. An algorithm (which makes some assumptions about the registers and instructions of the target machine) has been given in Aho *et al.* (1986). The problem with common subexpressions is that the results of these expressions have to be stored somewhere until they are used for the last time. This implies that after evaluating a certain subexpression, the number of registers in use does not always increase by exactly one (due to the additional register necessary to hold the value of that subexpression). It is possible that this increase is greater than one (if the cdag (connected dag; see below) contains CSEs that were not evaluated yet) or even smaller than one (if registers containing values of CSEs were used for the last time). Furthermore, the algorithm presented in Aho *et al.* (1986) cannot deal with values kept in registers at the beginning and at the end of a basic block. As with CSEs, such registers can be released as soon as their contents are not needed any more.

Before the modified algorithm (the algorithm and its proof can be found in Van Groningen (1990) that determines the evaluation order is explained some notions have to be introduced. A **(rooted) connected dag** (abbreviated as **cdag**) is defined as a dag that has a node r (the **root of the cdag**) such that all the other nodes of this dag are reachable from r. Further, let n be the number of cdags that have to be evaluated and g_i denote the ith cdag ($1 \le i \le n$). The evaluation order can be expressed by means of a permutation π of $1..n$ such that the cdag g_i is evaluated before a cdag g_j if $\pi^{-1}(i) < \pi^{-1}(j)$. Define $I(\pi, i)$ and $U(\pi, i)$ as:

$I(\pi, i)$ = the increase of the number of used registers due to the evaluation of $g_{\pi(i)}$ after evaluating $g_{\pi(1)}, \ldots, g_{\pi(i-1)}$.
$U(\pi, i)$ = the (additional) number of registers required for the evaluation of $g_{\pi(i)}$ (also after evaluating $g_{\pi(1)}, \ldots, g_{\pi(i-1)}$).

Note that $I(\pi, i)$ can be negative but $U(\pi, i)$ cannot. Furthermore, note that $U(\pi, i) \ge I(\pi, i)$.

Given an evaluation order π, the maximum number of registers used during the evaluation of $g_{\pi(i)}$ is

$$R(\pi, i) = U(\pi, i) + \sum_{k=1}^{i-1} I(\pi, k)$$

The number of registers necessary to evaluate all the graphs in an order specified by π is

$$R_m(\pi) = Maximum\ \{R(\pi, i) \mid 1 \le i \le n\}$$

Hence, finding an optimal evaluation order is equivalent to determining a permutation π_{min} such that for all other permutations π of $1..n$ it holds that $R_m(\pi_{min}) \le R_m(\pi)$.

A straightforward algorithm would generate all permutations π of $1..n$ and choose the one for which $R_m(\pi)$ is minimal. Unfortunately the complexity of such an algorithm is $O(n!)$ which is, of course, unacceptable.

The modified algorithm estimates the values of $I(\pi, i)$ and $U(\pi, i)$ beforehand (i.e. before determining the evaluation order) by respectively $I(i)$ and $U(i)$. It is required that the estimations are safe which implies that for all permutations π both $I(\pi, i) \le I(i)$ and $U(\pi, i) \le U(i)$ must be valid. After determining $I(i)$ and $U(i)$, the evaluation order of the cdags is given by the following two rules:

- first, evaluate all the cdags g_i with $I(i) \le 0$ from low to high $U(i)$;
- then, evaluate all the other cdags from high to low $D(i)$, where $D(i)$ is defined as $U(i)–I(i)$.

It will be clear that the cdags g_i with $I(i) \le 0$ have to be done first, for, after the evaluation of a graph with a non-positive I value, some registers may become free. To minimize the number of registers needed to evaluate the whole dag the cdags should be treated in ascending $U(i)$ order.

Why all the other cdags are ordered according to their D value is more difficult to see. This is illustrated with an example.

Suppose there are three graphs g_1, g_2, and g_3 with $I(1)=1$, $U(1)=2$, $I(2)=1$, $U(2)=2$, $I(3)=1$ and $U(3)=5$. Starting with g_3 only five registers are needed. Starting with one of the other two graphs at least six registers are necessary.

The two types of register of the MC68020 processor are not generally exchangeable. Furthermore, not all registers are freely available. Both facts make it necessary to adapt the algorithm described above. This is done in two stages. First, for each node of the dag it is decided whether this node is computed in an address register or in a data register. After that, for each cdag g_i two values of $I(i)$ are calculated: one for each type of register. Now, the total increase $I_t(i)$ is defined as:

$$I_t(i) = a * I_d(i) + d * I_a(i)$$

where a, I_a, d, I_d are resp. the number of address registers, the increase of address registers, the number of data registers and the increase of data registers. In the same $U_t(i)$ is defined as:

$$U_t(i) = a * U_d(i) + d * U_a(i)$$

The evaluation order is obtained by applying the previous algorithm using the functions I_t and U_t instead of I and U.

Consider again Figure 12.14: each node is supplied with two numbers, whereof the uppermost gives the I_d value of this node and the other number gives the U_d value. Since the I_a and U_a values do not matter they are omitted. The negative I values (of the `sub` and the uppermost register nodes) are a consequence of the fact that the contents of the register `d1` are not needed any more after evaluating these nodes. The result of the `mul` node can be stored in register d1 so, in contrast to the add node, no additional registers are needed to compute the result of this node.

The code generation phase

During the code generation phase, the dags of all the basic blocks are traversed. The order in which the cdags are treated is specified by the ordering phase. For each node of a cdag code is generated in a depth-first way: first, code is generated (recursively) for all of the arguments of a node (again in an order as specified by the ordering phase). Then the operation specified by the node is translated into one or more (pseudo-) MC68020 instructions assuming an unlimited number of (virtual) data and address registers.

The local register assignment phase

Finally, during the last phase of the translation real registers are assigned to the virtual registers allocated by the code generation phase. The algorithm that determines the evaluation order tries to minimize the number of data and address registers that are needed to evaluate a basic block. However, it may be the case that one of these numbers exceeds the number of registers that are actually available (i.e. the number of virtual registers exceeds the number of real registers). In that case it will be necessary to save the contents of one of the registers in memory in such a way that it can be used again. The problem is which register to choose. The strategy chosen takes the register of the required type whose contents will not be used for the longest time.

Consider Figure 12.14. During the last phase code is generated for this dag. The result is shown below:

```
F1:
muls.l   d1, d0
addq.l   #3, d1
sub.l    d1, d0
rts
```

12.6 Example of concrete code

In this section the quality of the code obtained by using the techniques described in this chapter is illustrated with the help of an example. For this purpose we take the Length example of Chapter 11.

```
:: Length !INT   ![x]      ->   INT;
   Length n      [ a | b ] ->   Length (+ n 1) b;
   Length n      [ ]       ->   n;
```

Consider the following Motorola code (some simplifications are made for reasons of clarity). When a function is called, the values on the bottom of a stack frame (see Chapter 11) are kept in registers. The registers a0–a1 are used for the bottom elements of the A-stack pointing to the root node to be overwritten and the first argument. The registers d0–d7 are used to transfer the bottom elements of the B-stack frame. Although the example above gives rise to many assembler instructions, the most important part of the resulting code that performs the recursive call of Length is a tight loop (between Length1 and Length2) that only takes a small number of instructions.

```
a_Length:                   ; The apply entry
                            ; a0 points to (List x)
                            ; a1 contains a pointer to (Length n)
   move.l  4(a1), a1        ; Fetch argument pointer
   move.l  (a1), a1         ; Fetch first argument n
   bsr     e_Length         ; Jsr to eval args entry
   subq.l  #2, fh           ; Reserve a node in the heap to store the result
   bcs     i_55             ; Branch to the garbage collector
i_56:                       ; Returning from the garbage collector
   move.l  hp, a0           ; Store the pointer to the new node
   move.l  #INT–DT, (hp)+   ; Store offset integer descriptor in the node
   move.l  d0, (hp)+        ; Store the integer result in the node
   rts                      ; rtn
n_Length:                   ; The node entry
                            ; a0 points to root (Length n (List x))
   lea     _cycle, a1       ; Store entry for cycle detection in a1
   move.l  a0, –(asp)       ; Save root pointer on A-stack
   move.l  a1, (a0)         ; Store cycle-in-spine entry in the root
   move.l  4(a0), a0        ; Fetch argument pointer of the root
   move.l  (a0)+, a1        ; Store first arg. (n) in a1
   move.l  (a0), a0         ; Store second arg. (List x) in a0
   bsr     e_Length         ; Jsr to eval args entry
   move.l  (asp)+, a0       ; Fetch root pointer from A-stack
   move.l  #INT–DT, (a0)    ; Store an integer descriptor in the node
   move.l  d0, 4(a0)        ; Store the integer result in the node
   rts                      ; rtn
e_Length:                   ; The eval args entry
                            ; a0 points to (List x)
                            ; a1 points to n
   move.l  (a0), d0         ; Fetch code/descriptor field second arg.
   bmi     e_0              ; Test on root normal form
   move.l  a1, –(asp)       ; Not in rnf, store first arg. on A-stack
   move.l  a0, –(asp)       ; Store second arg. on A-stack
   move.l  d0, a1           ; Set code pointer in a1
   jsr     (a1)             ; Evaluate second arg. to root normal form
```

```
    move.l  (asp)+, a0          ; Fetch second arg.
    move.l  (asp)+, a1          ; Fetch first arg.
e_0:                            ; Second arg. now in root normal form
    move.l  (a1), d0            ; Fetch code/descriptor field first arg.
    bmi     e_1                 ; Test on root normal form
    move.l  a0, –(asp)          ; Not in rnf, store second arg. on A-stack
    move.l  a1, a0              ; Root node always in a0
    move.l  d0, a1              ; Set code pointer in a1
    jsr     (a1)                ; Evaluate first arg. to root normal form
    move.l  a0, a1              ; Fetch first arg.
    move.l  (asp)+, a0          ; Fetch second arg.
e_1:                            ; First arg. now also in root normal form
    move.l  4(a1), d0           ; Store integer value n in d0
Length1:                        ; Entry for first rule alternative
                                ; a0 points to (List x)
                                ; d0 contains n
    cmp.w   #Cons+8–DT,2(a0)    ; Test whether list arg. is of type Cons a b
    bne     Length2             ; If not do second rule alternative
m_P1:                           ; First rule alternative is applicable
    move.l  a0, –(asp)          ; Save second arg. on A-stack
    move.l  4(a0), a0           ; Fetch argument pointer second arg.
    move.l  (a0)+, a1           ; Fetch pointer to a in a1
    move.l  (a0), a0            ; Fetch pointer to b in a0
    move.l  (a0), d1            ; Fetch code/descriptor field of b
    bmi     e_2                 ; Test on root normal form
    move.l  a1, –(asp)          ; Not in rnf, store pointer to a on A-stack
    move.l  a0, –(asp)          ; Store pointer to b on A-stack
    move.l  d0, (bsp)+          ; Store n on B-stack
    move.l  d1, a1              ; Set code pointer in a1
    jsr     (a1)                ; Evaluate to root normal form
    move.l  –(bsp), d0          ; Fetch n from B-stack
    move.l  (asp)+, a0          ; Fetch pointer to b in a0
    move.l  (asp)+, a1          ; Fetch pointer to a in a1
e_2:                            ; b in root normal form now
    addq.l  #1, d0              ; + n 1
    addq.l  #4, asp             ; Remove list argument
    bra     Length1             ; Recursion via a loop
Length2:                        ; Entry for second rule alternative
    cmp.w   #Nil+8–DT,2(a0)     ; Test whether list is a Nil
    bne     Length3             ; If not do additional rule alternative
m_P2:                           ; Second rule alternative is applicable
    rts                         ; Done, result in d0, rtn
Length3:                        ; Entry for additional alternative
    lea     l_1, a0             ; Load address of error message
    jsr     print               ; Print it out
    jmp     halt                ; Halted due to a mismatch
```

12.7 Performance

In this section the code generated using the optimization techniques presented in this chapter is compared with C (for a comparison of the execution speed offered by modern compilers for lazy functional languages we refer to Hartel and Langendoen (1993)). The gnu C compiler for the Sun3 is used, which generally gives faster code than the standard C compiler. It should be stated that, where possible, C has been used in an imperative way (i.e. using assignments and iteration instead of recursion where appropriate). The following test programs were used (see Smetsers *et al.*, 1991):

nfib	the well-known Nfib function with argument 30.
tak	the Takeuchi function, called with tak 24 16 8.
sieve	a program which generates the first 10 000 primes, using a quite optimal version of the sieve of Eratosthenes (outputs only the last one).
queens	counts all solutions for the (10) queens problem.
reverse	reverses a list of 3000 elements 3000 times.
twice	four times the twice function on the increment function.
rnfib	again the Nfib function, but now defined on reals, with argument 26.
fastfourier	the fast Fourier algorithm, on an array of 8K complex numbers. In the Concurrent Clean program a complex number is defined as a strict tuple of two reals.

Table 12.3 shows that recursive programs written in C appear to be slower than the ones written in Concurrent Clean. However, the iterative versions of the examples written in C are faster.

The test programs have the advantage that they are very small, such that a comparison can be made quickly. However, the question remains of how large 'real applications' written in a functional language behave compared to their imperative counterparts. Of course, it will depend highly on the kind of application. In general the functional program will run slower; not one or two orders of magnitude as was the case in the past, but some constant factor that depends on the kind of program. However, the loss of efficiency is already often quite acceptable. For example, a copy/paste editor written in Clean was, to our surprise, as fast as its imperative look-alike written in object-oriented C. It says something about both implementations. Anyhow, in comparison with the past, the difference in execution times between functional languages on the one hand and imperative languages on the other hand has significantly decreased.

Table 12.3 Performance measurements of the Concurrent Clean compiler version 0.7. The first three columns give the speed figures measured on different Macs using a 2Mb heap (with only 1Mb heap for the Mac+). All times are in seconds. The same programs are also tested on a SUN3/280, with a MC68020 processor running at 25 MHz, 2 Mb heap and compared with C. In Clean (!), strictness annotations are added by the programmer.

Program	*Clean* MacIIfx	*Clean* MacIIsi	*Clean* Mac+	*Clean* Sun3	*Clean (!)* Sun3	*C* Sun3
nfib	2.6	5.2	53	4.5	4.5	11
tak	2.6	5.3	53	4.9	4.9	11
sieve	4.4	9.4	260	8.1	6.8	4.5
queens	15	41	240	28	14	4.1
reverse	31	63	620	64	50	–
twice	0.86	1.8	Heap full	1.7	0.5	–
rnfib	6.1	13	2000	11	11	19
fastfourier	14	30	Heap full	34	19	9

The most worrying aspect is still the space consumption, which often is much too high for functional programs. Fortunately there are many more opportunities for efficiency improvement left: e.g. program transformations, sharing analysis, special code generation for UNQ-attributed objects, merging of B- and C-stacks etc. Also, the run-time system can be further improved. For instance, the current Clean system automatically switches between copying garbage collection (time-efficient) and mark–scan (space-efficient) to get the best of both worlds. All this will lead to much better efficiency in both time and space.

Summary

- The ABC machine can be realized in *hardware* or in *software*. A realization in hardware will be fast but such a machine will also be expensive. A software *interpreter* is easy to make, portable, but slow.
- A better performance can be obtained by a *code generator* that produces executable code from ABC instructions. To obtain a good performance one has to generate concrete target machine code. Such a code generator will probably be harder to port.
- Before code can be generated, *concrete data structures* have to be designed to represent the different memory components of the abstract ABC machine. Furthermore, one has to decide how to map these data structures onto the actual storages of the concrete architecture.

- In general the concrete data structures are quite similar to the abstract data structures. The main problem is caused by the fact that in the concrete case the available amount of memory is *limited*, so special actions are needed for memory management of the stacks (*boundary checks*) and the graph store (*garbage collection*).
- *Copying garbage collection* is very fast and a suitable method, but it needs a lot of memory. *Mark–scan garbage collection* can be used if there is not enough memory available for a copying collector. One can also implement both techniques and automatically switch between them. On-the-fly garbage collectors may be useful for real-time applications.
- Code generators try to use the available resources of the target machine as well as possible. Therefore one has to analyse whole sequences of ABC instructions (*basic blocks*). There are several optimization techniques possible to manipulate basic blocks: one can try to *change the flow of control* to minimize the unnecessary copying of information and to make an optimal use of the registers, one can try to *combine* boundary *checks* on stacks and on the heap, or replace any sequence of code by a more efficient one with the same effect.
- With the optimization techniques described in this part of the book it is possible to obtain efficient implementations of lazy functional languages on conventional architectures.

EXERCISES

12.1 Take the Concurrent Clean System and type in the `Length` function as defined in Section 12.6.

(a) Use the Clean System to generate ABC code. Compare this code with the code given in Section 11.3. What do you think is the purpose of the descriptors? How many stacks are there? What other kind of differences do you notice? Explain their purpose.

(b) Use the Clean System to generate assembly code. Compare the code with the code given in Section 12.6. Explain the differences.

12.2* Write the test programs of Section 12.7 in your favourite high-level functional language. Write the test programs in your favourite imperative language using an imperative programming style. Write the test programs in your favourite logical language. Measure the performance in both time and space of all programs. Compare the performance with the figures of Table 12.3. Explain the differences.