Chapter 11
Translating Clean into ABC code

This chapter discusses the transformation of Clean programs into ABC machine instructions (Figure 11.1). The relatively high-level functional language Clean is transformed into a low-level, imperative machine language. Hence, compared with the transformations described so far, the transformation from Clean into ABC code is the most complex one.

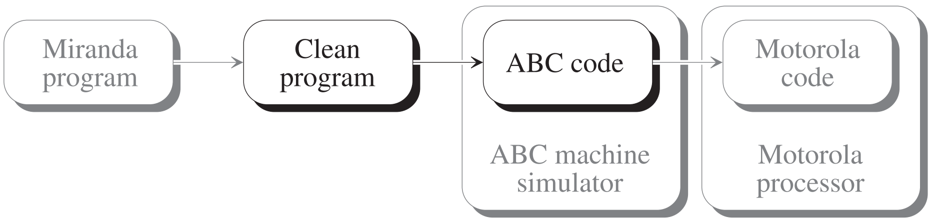


Figure 11.1 The second transformation step: Clean into ABC code.

The goal of the translation is, of course, to transform Clean programs into equivalent and efficient ABC programs. This ABC code is then further translated into instructions for the concrete target machine. The various target machines have quite different instruction sets, which makes it impossible to develop a compilation scheme to ABC code that is optimal for all possible concrete architectures. Nevertheless, several general optimizations can be used. These general optimizations consist of deviations from the standard operational semantics of Clean that are deliberately made to increase the performance of the generated code.

An important optimization is the following: although Clean graphs are mapped directly to graphs in the graph store of the ABC machine, this graph store is *not* updated after every rewrite step. During the rewriting the updating information is stored on the stacks and (implicitly) in the instruction sequence executed.

Another important optimization is induced by the fact that the performance of ABC programs heavily manipulating the graph store for simple computations will be low compared with ABC code that employs the B-stack whenever appropriate. So to achieve an efficient implementation of graph rewriting one should *avoid* the actual rewriting of the graph in the graph store where possible. For instance, in a straightforward scheme, subgraphs are constructed in the graph store to pass arguments from one function to another. But, when simple basic values are passed one does not need to construct these nodes at all: the values can be passed on the B-stack. The B-stack will be used in all cases where strict arguments of a basic type are passed to or returned by a function.

Section 11.1 informally describes graph rewriting on the ABC machine. Section 11.2 discusses the run-time system. Finally, Section 11.3 treats some high-level optimizations including the use of the B-stack.

11.1 Basic graph rewriting on the ABC machine

There are, in principle, many different ways to realize graph rewriting on the abstract ABC machine. This section discusses one straightforward method in an informal way. Only *pure* graph rewriting is considered. This means that only the graph rewriting part of the ABC machine is used. So the B-stack is not used in this section.

When a Clean program is translated into an ABC program, a sequence of ABC assembly instructions is generated for *each* rewrite rule and, in addition, for *each* rewrite rule alternative. In this code the functional reduction strategy and the graph rewriting operations are merged. For *each* Clean *rewrite rule* there is a sequence of ABC code to:

- construct a *stack frame* such that the arguments of the function can be accessed directly; furthermore, the arguments with global strict annotations are reduced to root normal form;
- handle the situation that none of the rule alternatives is applicable.

In addition, for *each* Clean *rule alternative* there is a sequence of ABC code to:

- determine whether the rule alternative matches the actual redex; if this is the case the corresponding rewrite is performed; if not, the next rule alternative is tried by executing its code.

Furthermore, there is a fixed piece of ABC code, called the **run-time system**, that

- initiates the reduction of a Start rule to *normal form*;

- *prints* the result of the computation on the input/output channel;
- *converts* curried applications into uncurried ones.

Entries (see Chapter 10) are important because they mark pieces of code that belong together. A sequence labelled with an entry can be executed by jumping to this entry (there are several jump instructions available). So entries can be called from outside. There are several kinds of entry generated in the ABC code. For each entry a certain **calling convention** is assumed: i.e. an interface between caller and callee. The different pieces of code, the entries and the calling conventions are described below.

11.1.1 Entries and calling conventions

One property of the functional strategy is that, once initiated, the reduction of a (sub)graph continues until it is in root normal form. This is realized as follows on the ABC machine. As was explained in the previous chapter, entries can be stored in the code field of a node. In general, the code field of a node will refer to a special entry, the **node entry**. When this node entry is called, the corresponding instructions will eventually reduce the subgraph rooted at that node to its root normal form (if it exists).

The reduction of a redex can be initiated by a jsr_eval instruction. This instruction calls the node entry stored in the code field of the root node of the redex and leaves a reference to this root on the top of the A-stack (see the definition of jsr_eval). When the node entry is entered, the root node is marked as being under reduction such that a cycle-in-spine can be detected. Since the ABC machine contains no tags, this marking is realized by overwriting the code field with a special entry (_cycle) that will produce an error message when it is called by a second attempt to reduce the same node.

Before the actual matching and rewriting can take place a stack frame is constructed on the A-stack. This **stack frame** contains:

- a reference to the root of the (sub)graph under reduction. This root is used as a **place-holder node** on the bottom of the stack frame. It will be updated with the root normal form when this is reached.
- the node-ids of all the arguments of the function that is called. The reference to the first argument is on top of the stack. The arguments that are marked as strict are in root normal form.

To construct the described stack frame, the arguments of the root node have to be pushed on the A-stack as well. Those arguments that are globally annotated as being strict are reduced to root normal form (again by using jsr_eval instructions). There is a special entry, called the

eval args entry (evaluate strict arguments), to reduce the corresponding arguments.

The actual matching and rewriting will take place depending on the actual rewrite rule alternatives (see Sections 11.1.2 and 11.1.3).

Consider the following rewrite rules:

```
:: F ![x]                  ![x]       ->   INT;
   F [ a | [ b | c ] ]   [ d | e ]  ->   G f f , f: [ b | e ];

:: G [x]  [x]   ->   INT;
   G a    b     ->   1;
```

Throughout this chapter, new parts in the pictures are printed bold. Graphs with unknown contents are drawn as grey boxes (Figure 11.2).

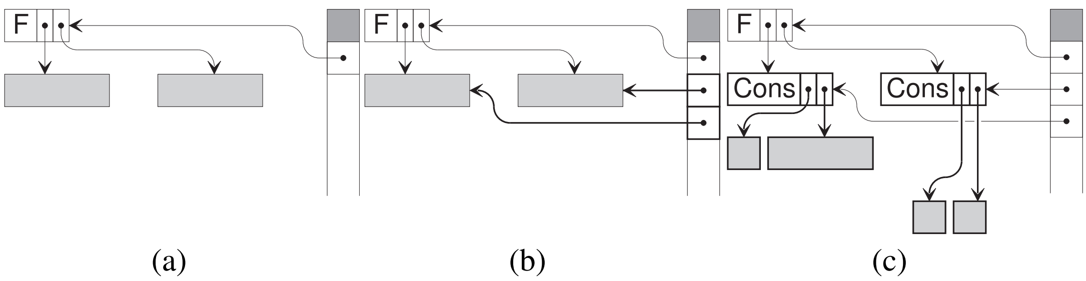


Figure 11.2 (a) The initial A-stack frame and graph just before entering the node entry. (b) The stack frame is extended with the node-ids of the arguments. (c) The stack frame as expected by the rule alternatives; both arguments are strict and hence both are reduced to root normal form.

For each rewrite rule alternative there is a separate **rule alternative entry**. Only the first rewrite alternative will be referenced outside the code for a particular rewrite rule. The code for each rule alternative entry consists of a *matching phase* which determines whether this alternative is applicable and a *rewriting phase* which performs the rewrite. When an alternative appears to be inapplicable, execution proceeds with the next one. Furthermore, for each rewrite rule, an **additional rule alternative entry** is generated that is executed when none of the rewrite rule alternatives is applicable. This means that a partial function has been called with a value on which it is not defined, which is generally considered as a run-time error (see Section 8.2.3). However, for well-typed partial functions it means the place-holder node can be updated with the root normal form reached.

A different situation arises when a *curried* function is used. As explained in Chapter 8, an automatic conversion has to convert a curried application into an uncurried one. For this purpose, a special internal

_AP-rule is defined in the run-time system (see Section 11.2). When sufficient arguments are supplied, the _AP-function will call the special **apply entry** of the rewrite rule. In this case, the node-id of the redex, the node-id of the last argument and a reference to the node containing the other arguments will be on the stack. To construct the same stack frame as described above, the last reference has to be replaced by the other arguments of the rewrite rule. Finally, the strict arguments have to be reduced to root normal form. So the eval args entry is called.

Summary of entries, calling conventions and their actions

Node entry

Called by: any function that needs the root normal form of the applied function.

Calling conventions:
- node-id of root of redex on top of A-stack.

Actions:
- mark root node redex for cycle-in-spine detection; this node now serves as place-holder node;
- push the node-ids of all the arguments of the redex on the A-stack; last argument on top of the stack;
- proceed with eval args entry.

Eval args entry

Called by: node entry as well as apply entry.

Calling conventions:
- node-id of place-holder node on the stack;
- node-ids of all arguments of the function called above it, node-id of first argument on top of A-stack.

Actions:
- reduce those arguments that are marked as strict to root normal form;
- proceed with first rule alternative entry.

Rule alternative entry

Called by: the first rule alternative can be called by any function; the other rule alternatives are called by the previous one when they do not match.

Calling conventions:
- the stack frame is complete: node-id of place-holder node on the stack, node-ids of arguments on top of it, strict arguments are in root normal form.

Actions:
- try to match this alternative;
- if no match, proceed with next rule alternative entry;
- if match, evaluate strict nodes of right-hand side and construct contractum;
- when a root normal form is reached, the place-holder node is overwritten with the contractum; return to caller;
- otherwise: build stack frame for root of right-hand side;

- proceed with the first rule alternative entry of the root of right-hand side.

Additional rule alternative entry

Called by: last rule alternative entry.

Calling conventions:
- a complete stack frame.

Actions:
- in general: generate an error message and halt the program;
- for well-typed partial functions: fill place-holder node with root normal form.

Apply entry

Called by: built-in _AP-function when a curried application has enough arguments.

Calling conventions:
- node-id of place-holder node on A-stack;
- node-id of node containing $n - 1$ arguments on top of it;
- node-id of last argument on top of A-stack.

Actions:
- pop the two node-ids on top and push the node-ids of all the arguments of the function called on the A-stack;
- proceed with eval args entry.

11.1.2 Matching

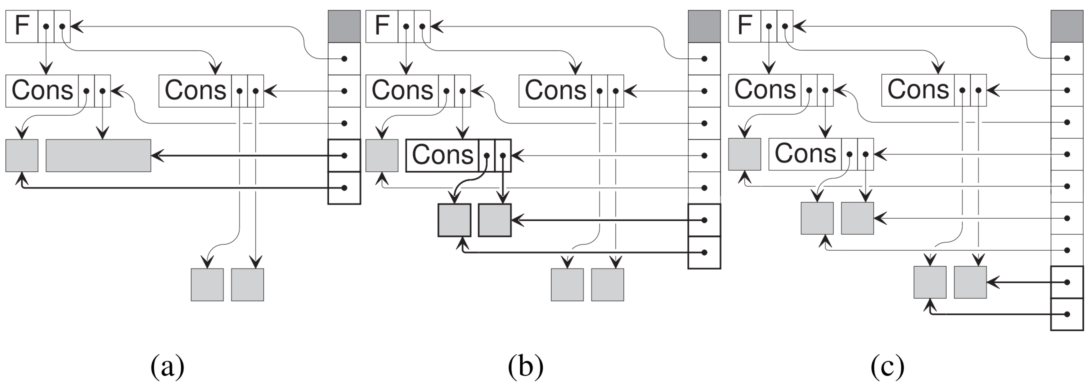


Figure 11.3 (a) The situation after matching the top level of the first argument and pushing its subarguments. (b) The second subargument of the first argument is reduced and matched. After the successful match the node-ids of the subarguments are pushed on the stack. (c) The second argument also matches. So this rule alternative is applicable.

In each rewrite rule alternative the arguments are matched from left to right. If the formal argument in the corresponding rewrite rule is a variable the argument trivially matches. Otherwise, when the actual argument is not known to be in root normal form already, it is brought in root normal form by calling its node entry. Now the symbol in the graph can be compared with the symbol specified in the redex pattern. When the symbols match (are equal), its subarguments are pushed on the A-stack as well (Figure 11.3). Subpatterns are matched in the same way. In this way all node-ids that appear on the left-hand side of the rewrite rule alternative are pushed on the A-stack. The advantage of this is that these node-ids and hence the corresponding nodes become directly accessible in the rewriting phase.

As soon as an argument is found that does not match the corresponding redex pattern, all subarguments are popped from the stack and execution proceeds with matching the next rule alternative. When the matching of all arguments succeeds the rewrite must be performed.

11.1.3 Rewriting

When the redex matches a rewrite rule alternative, the rewriting can take place. According to the semantics of GRSs, the contractum must be constructed (if present in the rule) and all references to the root of the redex must be redirected to the root of the contractum. During the matching phase all node-ids that appear on the left-hand side of the rewrite rule alternative have already been pushed on the A-stack such that they are directly accessible in the rewriting phase.

To increase the performance of the generated code some deviations of the standard GRS semantics are made.

As explained in Section 8.3.2, before reducing the root node of the right-hand side all strict nodes other than the root node are evaluated to root normal form. However, not the whole contractum is built before these strict nodes are evaluated. Only those parts of the contractum that are necessary for this evaluation (lazy subgraphs, sharing, cycles) are built before this evaluation is started.

Furthermore, as already mentioned in Section 10.2, it is very inefficient to examine the whole graph and replace every reference to the root of a redex by a reference to the contractum root. A more efficient implementation is achieved by *overwriting* the root of the redex with the root of the contractum. In this way all references to the redex are automatically redirected to the contractum.

Finally, the graph store is *not* updated after each rewrite step. It is only updated when a root normal form is reached. During the rewriting the information is stored on the stacks and (implicitly) in the instruction sequence executed. Three situations are distinguished: the contractum is *not known to be in root normal form*; the specified contractum is a *root normal form;* and *no contractum* is specified in this rule alternative.

Redirection to a reducible contractum

When the contractum is not known to be in root normal form, the rewrite rule associated with the symbol in this root must be applied. It makes no sense to fill the redex with the root of the contractum. The new rewrite rule would unpack this node immediately since the functional strategy will indicate this redex as the one to rewrite. So it is possible and more efficient to delay the update of the graph store until a root normal form is reached.

Hence, when the reduction of a new subgraph is demanded, the corresponding root of the redex is pushed once on the bottom of the stack frame. In the case that the reduction continues with the reduction of the contractum, a new stack frame is constructed for the first rule alternative of the corresponding rewrite rule. Rewriting proceeds immediately with that first alternative.

When the right-hand side root function is called, the current function does not need its own stack frame any more. So the space occupied by the old stack frame can be reused for building the new one. The original redex stays as place-holder on the bottom of the new stack frame such that it can be overwritten when the root normal form is reached. The node-ids of the old arguments are removed from the stack and the new ones are pushed instead. After that, a jump (and not a jsr_eval) is performed to the first rule alternative of the called function. In this way *tail recursion* is removed automatically.

Redirection to a contractum in root normal form

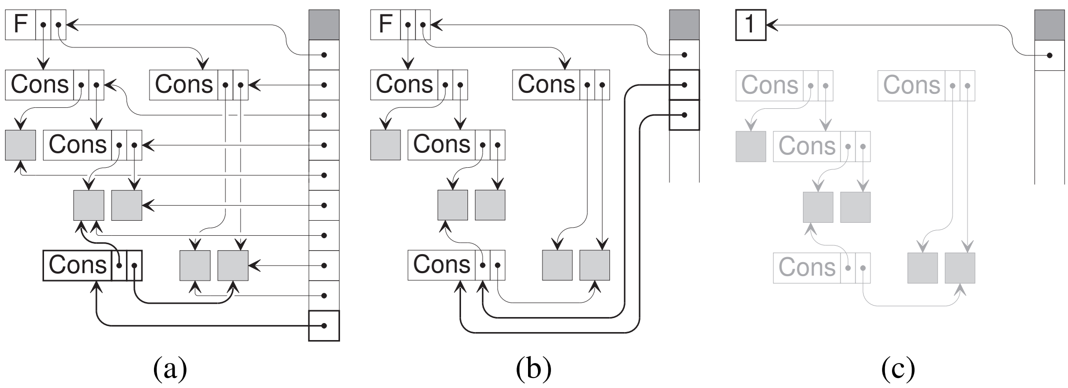


Figure 11.4 (a) The shared node, f, is constructed in the graph store. (b) The stack frame is ready for matching the first rule alternative of G. (c) The reduction according to rule G has been performed, a root normal form is reached; garbage nodes are drawn dotted.

The ABC machine has variable-sized nodes. This means that it is always possible to overwrite a node with arbitrary new contents, even if the new node is larger than the old one. So when the contractum is in root normal form, the place-holder node at the bottom of the stack frame is overwritten with the root of the contractum. In this way all references to the redex are automatically redirected to the contractum: a very cheap and commonly used implementation of redirection.

For a machine without variable-sized nodes the place-holder node is usually overwritten by an **indirection node**: a special node that fits in any node which points to the actual node. A drawback of indirection nodes is that they make an implementation a bit more complicated and inefficient because these indirection nodes can appear anywhere in a graph. So special attention has to be paid to deal with such nodes.

Redirection to an existing graph

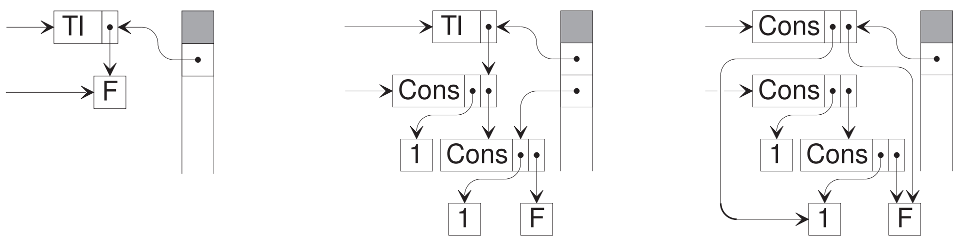
When the right-hand side of a rewrite rule only consists of a redirection, there is a problem. The root of the redex has to be redirected to the root of an existing graph somewhere in the graph store. Introduction of indirection nodes may seem unavoidable.

However, the following approach is taken. After the redirection to some existing (sub)graph, the functional strategy will continue with the reduction of that (sub)graph to root normal form. The idea is now to postpone the redirection until after the reduction of the indicated subgraph (see Figure 11.5).

To illustrate these graph manipulations an example is shown .

```
::  Tl ![x]       ->    [x];
    Tl [ h | t ]  ->    t;

::  F  ->    [INT];
    F  ->    [ 1 | F ];
```



(a) (b) (c)

Figure 11.5 (a) Start of the reduction for the graph Tl F. State at the node entry. The arrows from outside indicate sharing. (b) After the reduction of the redex t, before the rewriting. (c) After copying the top node. Both nodes contain the proper root normal form.

This means that first the reduction to root normal form of the redex indicated by the redirection is started. When the root normal form is reached, the root node of this redex will be overwritten as usual. Next, the postponed redirection is performed. This is implemented by overwriting the place-holder node with a *copy* of the root node of the root normal form obtained.

This optimization results in a graph that is *different* from the one that should be obtained according to standard GRS semantics. The redirected arcs point to a copy of the root node instead of to the root node itself. Fortunately, both the graphs remain equal modulo unravelling (see also Chapter 14).

11.1.4 Example of the rewriting process

Consider the following Clean rewrite rule to compute the length of a list:

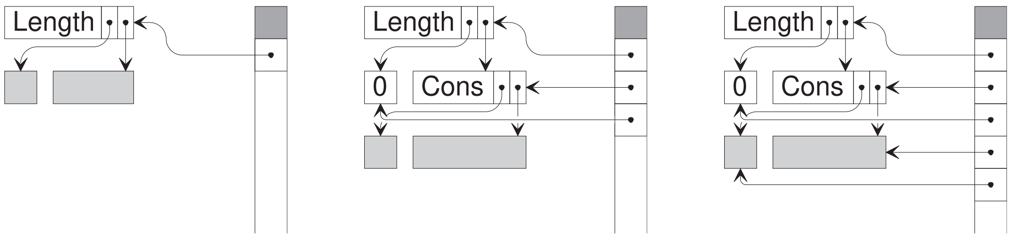
```
:: Length !INT   ![x]      ->   INT;
   Length n      [ a | b ] ->   Length (+ n 1) b;
   Length n      [ ]       ->   n;
```

The corresponding ABC code using the described calling conventions is:

```
[            Descriptor
"Length"     "a_Length" 2 "Length"      ,  || The generated descriptor
             Label                         || The apply entry
"a_Length",Repl_args 1 1                ,  || Prepare the stack frame
             Jmp "e_Length"             ,  || Start reducing strict arguments
             Label                         || The node entry
"n_Length",Set_entry "_cycle" 0         ,  || Mark node to detect cycle-in-spine
             Push_args 0 2 2            ,  || Push the arguments
             Label                         || The eval args entry
"e_Length",Jsr_eval                     ,  || Reduce first argument to rnf
             Push_a 1                   ,  || Copy 2nd argument on top stack
             Jsr_eval                   ,  || Reduce it to rnf
             Pop_a 1                    ,  || Pop duplicated second argument
             Label                         || Entry for first rule alternative
"Length1",   Eq_desc_arity "Cons" 2 1,     || Match second argument
             Jmp_false "Length2"        ,  || Goto next alternative if match fails
             Push_args 1 2 2            ,  || Push subarguments
             Push_a 1                   ,  || Rewrite according to alternative 1
             Jsr_eval                   ,  || Reduce b
             Create                     ,  || Node for result of + n 1
             Create                     ,  || Node for 1
             FillI 1 0                  ,  || Fill this node
             Push_a 5                   ,  || Push n; it is known to be a rnf
```

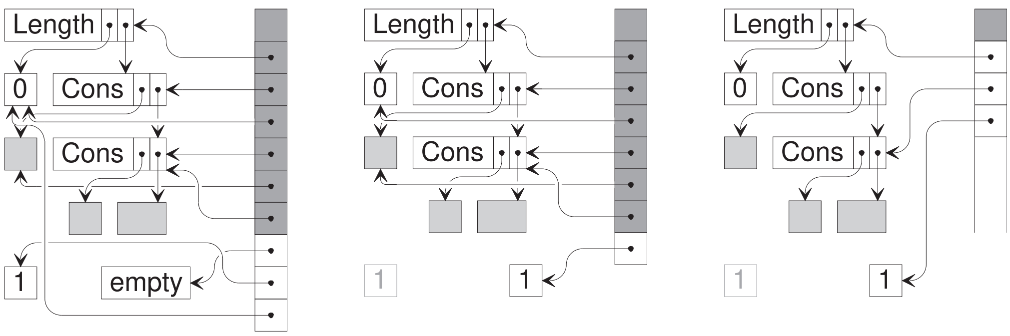
```
            Jsr "+1"                 ,  || Reduction of + n 1
            Update_a 1 5             ,  || Adapt stack frame to call Length
            Update_a 0 4             ,  ||
            Pop_a 4                  ,  || Remove old (sub)arguments
            Jmp "Length1"            ,  || Goto alternative 1 of Length
            Label                       || Entry for second rule alternative
"Length2",  Eq_desc_arity "Nil" 0 1  ,  || Match argument
            Jmp_false "Length3"      ,  || Goto next alternative if match fails
            Fill_a 0 2               ,  || Rewrite according to alternative 2
                                        || n known to be a rnf, so it is copied
            Pop_a 2                  ,  || Remove arguments from stack
            Rtn                      ,  || Rnf reached
            Label                       || Entry for additional alternative
 "Length3", Jmp "_mismatch"          ]  || There is a mismatch
```

The machine state during reduction of this code for the graph Length 0 [1, 2] is shown in Figure 11.6:



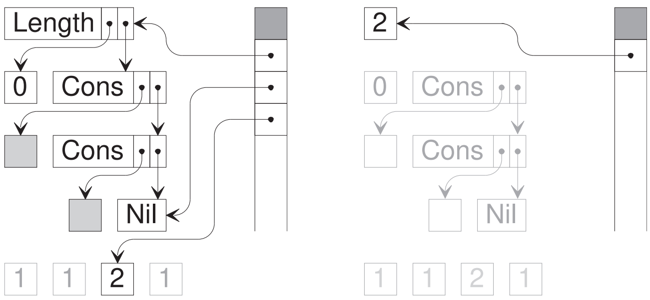
(a) (b) (c)

Figure 11.6 (a) Initial state at node entry. (b) Initial state at first alternative entry. (c) After successful match of first rule alternative.



(d) (e) (f)

(d) Reduction according to the first rule alternative. First, the last strict argument is reduced. The state before calling + 1 to reduce the first strict argument is shown. (e) State returned by + 1. Garbage nodes are displayed dotted. (f) State before calling Length1 recursively. (*continued overleaf*)



(g) (h)

Figure 11.6 (*continued*) (g) State before last call of Length1. (h) State of ABC machine after reaching the rnf.

11.2 The run-time system

Each Clean rewrite rule is translated into a sequence of ABC instructions. To let the machine perform graph rewriting properly, some additional ABC instructions also have to be defined that are independent of a concrete program. This fixed piece of ABC code is called the **run-time system**. The main purpose of the run-time system is to force evaluation to normal form, to perform I/O, and to convert curried functions.

Reduction to normal form and printing

The compilation scheme explained above produces code such that a redex is reduced to root normal form. The run-time system contains the routine init_graph to build the initial redex, the Start node. Then, it initiates the reduction of this Start node to *normal form*. The reduction to normal form and the printing of subgraphs in normal form are done by the routine _driver. The Start node is reduced to normal form as follows: first the Start node is reduced to root normal form, the symbol in the resulting root node is printed and the driver process recursively continues with the arguments of the root normal form thus obtained.

```
[               Label                          || The initiator of graph reduction
"init_graph",   Create                      ,  || Create and fill the start node
                Fill "Start" 0 "n_Start" 0  ,
                Jsr "_driver"               ,  || Print the nf of start node
                Print "\n"                  ,  || Print a newline
                Halt                        ,  || The program is finished
                Label                          || The global print driving routine
"_driver",      PushI 0                     ,  || Closing bracket count
                Label                          || Label for tail recursion
"_print",       Jsr_eval                    ,  || Reduce top node to rnf
                Get_node_arity 0            ,  || Get number of args in node
                EqI_b 0 0                   ,  || No arguments ?
```

```
              Jmp_false "_args"        ,
              Label                      || Print last argument
"_print_last", Print_symbol 0          , || Do it
              Pop_a 1                  , || Remove node
              Pop_b 1                  , || Remove arity
              Label                      || Print the closing brackets
"_brackets",  Eql_b 0 0                , || Bracket count equal 0 ?
              Jmp_true "_exit"         , || Yes; finished
              Print ")"                , || No; print bracket
              Decl                     , || Decrement bracket count
              Jmp "_brackets"          , || Next bracket
              Label                      || Finished with this node
"_exit",      Pop_b 1                  , || Remove bracket count
              Rtn                      ,
              Label                      || Start the printing of arguments
"_args",      Print "("                , || An opening bracket
              Print_symbol 0           , || The symbol of the node
              Get_desc_arity 0         , || Arity corresponding to symbol
              Repl_args_b              , || Replace node by its arguments
              Pop_b 1                  , || Pop descriptor arity
              Label                      || Loop to print arguments
"_arg_loop",  Print " "                , || Space between elements
              Eql_b 1 0                , || Last argument ?
              Jmp_false "_next_arg"    ,
              Pop_b 1                  , || Remove arg counter
              Incl                     , || Increment bracket counter
              Jmp "_print"             , || Optimized tail recursion
              Label                      || Print an arg; not the last one
"_next_arg",  Jsr "_driver"            , || Recursion to print argument
              Decl                     , || Decr arg cnt; driver removes arg
              Jmp "_arg_loop"          ] || Next argument
```

Miscellaneous definitions in the run-time system

The run-time system contains some special entries for general use and standard descriptors for elements of basic types. As in the previous chapter, only integer and Boolean values are considered here.

The _rnf entry is used as node entry for graphs in root normal form. The _cycle entry is stored upon entrance of a node entry when the reduction of the subgraph rooted with this node is started. In this way a cycle-in-spine error is detected when such a node is revisited. The _mismatch entry is used when none of the rule alternatives is applicable.

```
[             Descriptor
"INT"         "_rnf" 0 "integer"       , || Reserved first descriptor
              Descriptor
```

```
"BOOL"      "_rnf" 0 "boolean"          ,  || Reserved second descriptor
            Label                          || Node entry for graphs rnf
"_rnf",     Rtn                         ,  || Return immediately
            Label                          || Node entry to detect cycles
"_cycle",   Print "Cycle-in-spine: Reduction interrupted",
            Halt                        ,  || Stop the reduction
            Label                          || Exit when no alt. applicable!
"_mismatch", Print "Mismatch: Reduction interrupted",
            Halt                        ]  || Stop the reduction
```

Curried functions

To handle curried function applications, the run-time system contains a special predefined rule _AP. The generic _AP rule converts a curried symbol into its uncurried form as soon as all its arguments are available. To obtain this effect all occurrences of curried symbols in Clean are transformed into applications of the function _AP. If F is defined with arity *n*, say F a_1 a_2 ... a_n -> a_r, then _AP (... (_AP (_AP F a_1) a_2) ...) a_n reduces to F a_1 a_2 ... a_n (see Section 8.1.6).

So the call of a curried function is actually a call of the function _AP. This function is strict in its first argument: the function that has to be applied. Each _AP node provides one additional argument to the function. If there are too few arguments the expression is in root normal form. When all arguments are there the indicated function is called.

On the ABC machine the arguments of the curried function are collected one by one in the corresponding apply node. When the last argument is supplied and all other required arguments are already collected, a stack frame according to the apply entry is constructed and execution continues on that entry of the corresponding function. Otherwise, the next argument is collected on its turn and a root normal form is reached. The code for _AP is similar to the code for other functions.

```
[           Descriptor
"_AP"       "n_AP" 2 "_AP"
            Label                          || The node entry for _AP
"n_AP",     Set_entry "_cycle" 0        ,  || Mark node as being reduced
            Push_args 0 2 2             ,  || Push the function and its arg
            Jsr_eval                    ,  || Reduce curried function to rnf
            Label                          || Entry for rule alternative _AP
"_AP1",     Get_node_arity 0            ,  || Number of args in node
            Get_desc_arity 0            ,  || Number of args for symbol
            SubI                        ,  || Number of arguments needed
            EqI_b 1 0                   ,  || Last argument supplied?
            Jmp_false "yet_args_needed",   || No; update node and return
            Pop_b 1                     ,  || Yes; start reduction of function
            Push_ap_entry 0             ,  || Push apply entry on C-stack
```

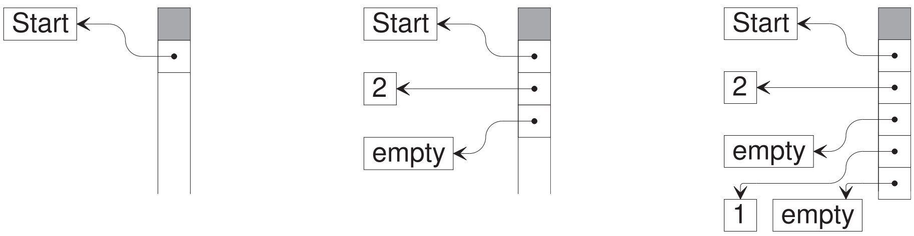
```
                 Rtn                              ,  || Jump to that entry
                 Label                               || _AP doesn't supply last arg
"yet_args_needed"                                 ,  || Update apply node and return
                 Push_a 1                         ,  || Argument supplied here
                 Add_args 1 1 3                   ,  || Update apply node
                 Pop_a 2                          ,  || Clean up stacks
                 Pop_b 1                          ,  ||
                 Rtn                              ]  || Rnf reached
```

This is illustrated by the following example (Figure 11.7):

```
Start ->    Inc 2;
Inc   ->    Plus 1;
Plus  ->    +;
```

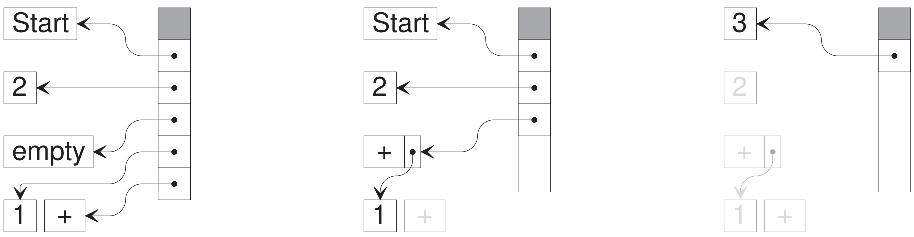
which is internally transformed into:

```
Start ->    _AP Inc 2;
Inc   ->    _AP Plus 1;
Plus  ->    +;
```



(a) (b) (c)

Figure 11.7 (a) Initial state. Reduction starts at the node entry of Start. (b) Creation of the arguments of _AP. Second argument is built. First argument is strict. (c) Reduction according to Inc. Second argument is created. First argument is strict.



(d) (e) (f)

(d) Reduction according to Plus. The redex node is filled with an rnf. (e) + has not enough arguments to become a function. So the node is filled with the rnf. (f) After reduction according to + the top node is replaced by its rnf.

11.3 Optimizations

The code generated by the compilation scheme presented above can be improved at many places. The most important improvement is to use the B-stack instead of the nodes in the graph to pass basic values between functions. The use of the B-stack will be described informally and illustrated with an example. Afterwards some other high-level optimizations will be discussed.

11.3.1 The extended compilation scheme using the B-stack

Basic values are manipulated on the B-stack of the ABC machine. This implies that every computation involving basic values requires the transportation of the values to the B-stack and the shipment of the result back to a node in the graph store. When the result is used again as argument in another computation there is too much data transportation. To reduce the unnecessary movement of data, the compilation scheme must be changed such that basic values stay on the B-stack as much as possible. To achieve this, the stack frame expected by the rewrite alternatives is changed: strict arguments of *simple basic type* (integers, reals, characters, booleans, files) are passed on the B-stack and the result of a reduction is also left on the B-stack when it is of such a basic type.

The calling conventions for the node entry and the apply entry remain unchanged, but the code corresponding to the eval args entry has to be changed since it has to take care of the transport of basic values between the graph and the B-stack. After the reduction of a strict argument of a simple basic type, the value contained in the node in the graph store is copied to the B-stack. When the result of the function is also of a basic type the first rule alternative entry is called as a subroutine. The corresponding code will yield the root normal form on the B-stack. Returning from the subroutine call the result is stored in the place-holder node.

Summary of changed calling conventions and their actions

Eval args entry

Called by: node entry as well as apply entry.

Calling conventions:
- node-id of place-holder node on the stack;
- node-ids of all arguments of the function called above it, node-id of first argument on top of A-stack.

Actions:
- reduce those arguments that are marked as strict to root normal form;
- when the strict arguments are of a simple basic type, the values are pushed on the B-stack and the corresponding node-ids are popped;

- proceed with first rule alternative entry, but do this via a jsr when the result is of simple basic type in order to come back to copy this result from the B-stack back to the A-stack.

Rule alternative entry

Called by: the first rule alternative can be called by any function;
- the other rule alternatives are called by the previous one when they do not match.

Calling conventions:
- the stack frame is complete: node-id of place-holder node on the stack, node-ids of arguments on top of it, strict arguments are in root normal form, strict arguments of complex type on A-stack, the others on the B-stack.

Actions:
- try to match this alternative;
- if no match, proceed with next rule alternative entry;
- if match, evaluate strict nodes of right-hand side and construct contractum;
- when a root normal form is reached, the place-holder node is overwritten with the contractum; return to caller;
- otherwise: build stack frame for root of right-hand side;
- proceed with the first rule alternative entry of the root of right-hand side.

The complexity of the code generation is increased significantly by this new calling convention. Arguments and results must be moved to the desired place at every occurrence. This is not difficult, but it involves an elaborated case analysis.

An example of the use of the B-stack

To show the changes resulting from the new calling conventions the ABC code for the Length example of Section 11.1.4 is shown.

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

Both arguments are strict. Note that the first argument is also a basic value, so it will be passed on the B-stack to the rule alternatives. They will also store the result on the B-stack. The integer addition used on the right-hand side also expects strict basic values as argument and it delivers a strict basic value as well. So no nodes have to be created when + is called, and all values are passed directly on the B-stack. The same holds for the first argument of Length when it calls itself recursively.

```
[           Descriptor
"Length"    "a_Length" 2 "Length"     ,  || The generated descriptor
            Label                        || The apply entry
"a_Length",Repl_args 1 1              ,  || Prepare the stack frame
```

```
              Jmp "e_Length"             ,  || Start reducing strict arguments
              Label                         || The node entry
"n_Length",Set_entry "_cycle" 0          ,  || Mark node to detect cycle-in-spine
              Push_args 0 2 2            ,  || Push the arguments
              Label                         || The eval args entry
"e_Length",Jsr_eval                      ,  || Reduce first argument to rnf
              PushI_a 0                  ,  || Copy first argument to the B-stack
              Pop_a 1                    ,  || Pop first argument from A-stack
              Jsr_eval                   ,  || Reduce second argument to rnf
              Pop_a 1                    ,  || Pop duplicated second argument
              Jsr "Length1"              ,  || Initiate rewriting by alternatives
              FillI_b 0 0                ,  || Fill node with result of reduction
              Pop_b 1                    ,  || Pop result of the reduction
              Rtn                        ,  || Done; node is in rnf
              Label                         || Entry for first rule alternative
"Length1",  Eq_desc_arity "Cons" 2 0,       || Match argument
              Jmp_false "Length2"        ,  || Goto next alternative if match fails
              Push_args 0 2 2            ,  || Push subarguments
              Push_a 1                   ,  || Rewrite according to alternative 1
              Jsr_eval                   ,  || Reduce b
              PushI 1                    ,  || Second argument for +
              Push_b 1                   ,  || First argument for +
              Jsr "+1"                   ,  || Reduction of + n 1
              Update_a 0 3               ,  || Adapt A-stack frame to call Length
              Update_b 0 1               ,  || Adapt B-stack frame to call Length
              Pop_a 3                    ,  || Remove old args from A-stack
              Pop_b 1                    ,  || Remove old args from B-stack
              Jmp "Length1"              ,  || Goto alternative 1 of Length
              Label                         || Entry for second rule alternative
"Length2",  Eq_desc_arity "Nil" 0 0      ,  || Match argument
              Jmp_false "Length3"        ,  || Goto alternative 3 if match fails
              Pop_a 1                    ,  || Remove arg
              Rtn                        ,  || Rnf reached
              Label                         || Entry for additional alternative
"Length3",  Jmp "_mismatch"              ]  || There is a mismatch
```

In Figure 11.8 snapshots of the reduction process are shown (compare them with the pictures in Figure 11.6). The efficiency gained by this optimized compilation scheme depends on the rules to transform and the implementation of the ABC machine. The measured efficiency gain for the length rule given as an example using a reasonable implementation is about a factor of 2.5 when garbage collection is excluded (see Chapter 12 for more information on garbage collection). However, a lot of garbage collection is prevented as well. The efficiency gained by eliminating the garbage collection depends heavily upon the circumstances. Including the garbage collection the efficiency gain of the length function is typically about a factor of 6.5.

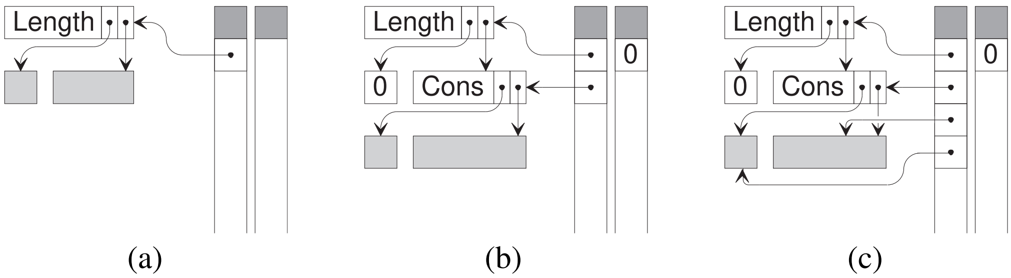
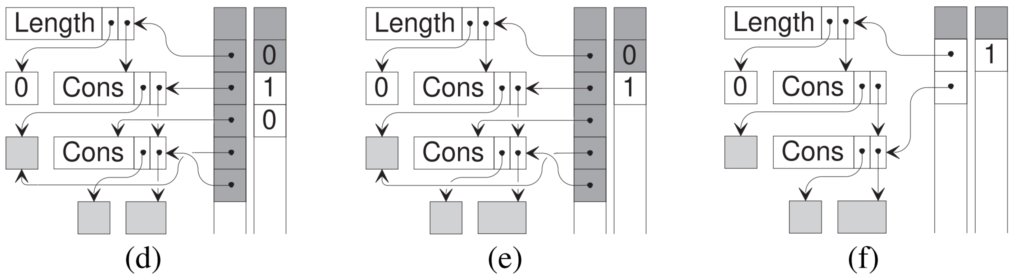
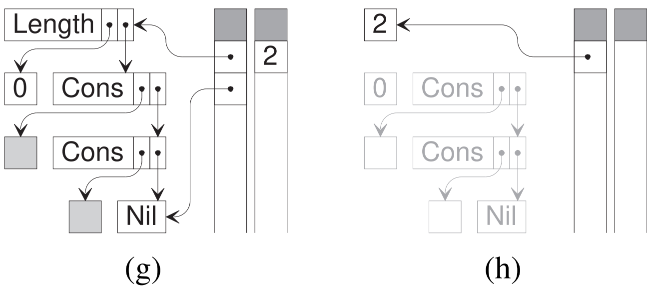


Figure 11.8 (a) Initial state at node entry. (b) State at first rewrite alternative entry. (c) After successful match of first rule alternative.



(d) Reduction according to the first rule alternative. State before calling +1. (e) State returned by +1. (f) State before calling Length1 recursively.



(g) State before last call of Length1. (h) State of ABC machine after the rnf is reached.

The example above shows the successful use of the B-stack. It included very little overhead involving the movement of data between the graph and B-stack. This is not always the case. It is possible to construct an example such that arguments just happen to be expected on the other stack. Projection functions, like the identity function, always work on the A-stack. This induces an overhead when they are called with arguments that live on the B-stack. A solution for this problem will be discussed in the next section.

Unnecessary overhead may occur when using the B-stack:

```
:: I !x    ->    x;
   I x     ->    x;

:: F !INT !INT    ->    INT;
   F a b          ->    + (I a) b;
```

Both arguments and the result of function F will be passed on the B-stack by the rule alternative entry. But, unfortunately, the identity function expects its polymorphic typed argument on the A-stack and also leaves its result there. So a has to be put back in a node to pass it to I and the result must again be extracted from a node to pass it to +.

11.3.2 Other optimizations

The code generated by the schemes above is generally more efficient than the scheme in Figure 11.6, but can be improved much further. Here, we mention some of the other optimizations that are possible. Many of them are implemented in the Concurrent Clean System (Smetsers, 1989).

- The elements of strict tuples can also be passed on the stacks instead of via a reference to a node in the graph. In addition, it is possible in Clean to put strict annotations in a tuple type appearing in a strict context (see Section 8.3). When these arguments are of a simple basic type they can also be passed on the B-stack instead of the A-stack. So in that case there is not much difference in terms of efficiency between a function with more than one argument and a function that carries its arguments in a strict tuple. When a function yields a tuple as result, the tuple arguments are returned via the stacks as well. Functions that return a tuple of which the basic typed arguments are marked as being strict can return the corresponding values via the B-stack.
- When a new stack frame is constructed over the old the references on the stack have to be rearranged in the right order. A simple solution uses temporary variables to construct the new frame before the old one is overwritten. A more clever algorithm will try to minimize the number of moves (updates) and additional amount of space needed for the construction of the new frame.
- As shown above, there are some cases where arguments are shipped from A-stack to B-stack and backwards frequently. The calling conventions for such functions can be changed such that they can handle items on both stacks. This requires, however, a number of entries that is exponential with respect to the number of

arguments since all cases have to be covered, so this will only be practical for functions with a very small number of arguments.

- For projection functions for which the right-hand side consists of a single node-id, the place-holder node of the function application can be used, so a new place-holder node need not be created. In this way one prevents a possible series of place-holder nodes to which results are copied.
- It is very important to use in-line code substitution for very short functions. Instead of performing a subroutine jump, the body of the routine is substituted in place. This makes it possible, for instance, to generate the `AddI` instruction instead of a `Jsr +1`.
- In the present compilation schemes the matching of each rule alternative starts from scratch. However, the use of information from the previous alternative and type information can speed up the matching process considerably. So it is better to use a finite state machine for the matching of the rule alternatives instead of starting all over again for each alternative.
- Constant graphs which occur on the right-hand side of rewrite rules need not be built each time the rule is used. These graphs can be created once. The node-ids of the root of these graphs can be used instead of building a new one at each occurrence.
- One can use scratch nodes for nodes which will be used once for a very short time. This occurs when a function delivers its result in the graph which is required on the B-stack; the result is stored in the node, moved to the B-stack and the node becomes garbage immediately.
- It is important that nodes are represented as compactly as possible. Special compact representations can be chosen for nodes containing basic values or nodes with few arguments.
- One can improve the performance of predefined functions (δ-rules) by hand-coding them in assembler.
- One can reuse `UNQ`-attributed function arguments (see Section 8.5) for the construction of the function result. This makes it, for example, possible to define a traditional sort function on a list with a unique spine. Sorting then can take place *in-situ*: no new Cons nodes have to be constructed (Smetsers *et al.*, 1993).

Besides these optimizations in the area of code generation for the ABC machine there are also important optimizations possible on the level of the Clean language itself: Clean program transformations. Functions can be partially evaluated at compile-time. Clearly, this is possible for expressions that consist only of constants. One can create specialized function calls for certain function applications in which it is, for example, clear that one or more of the alternatives of the original function will never match, etc. Of course, the transformations must not change the termination behaviour of the program.

Another important optimization on the level of Clean itself concerns the analysis of *all* applications of a function. This can, for instance, greatly improve the strictness analysis of Chapter 7, in which only the function definition is taken into account.

11.3.3 Efficiency gain of the optimizations

To analyse the effect of a subset of the optimizations mentioned a measurement of the performance of the well known Nfib function is presented. The integer delivered by the Nfib function is the number of performed function calls. The Nfib-number is obtained by dividing this number by the execution time. So the **Nfib-number** is the number of function calls per second that is executed for the calculation of the result in this particular example. The Nfib 'benchmark' is often used as a measure for the basic efficiency of an implementation of a functional language. Although it gives some indication of the speed offered, it is not at all a representative benchmark. The Nfib-number is extremely sensitive to the optimizations discussed. It actually gives the maximum number of function calls per second under ideal circumstances.

```
:: Nfib !INT  ->  INT;
   Nfib 0     ->  1;
   Nfib 1     ->  1;
   Nfib n     ->  ++ (+ (Nfib (-- n)) (Nfib (- n 2)));
```

The table below gives an idea about the gain in efficiency obtained by the optimizations mentioned. Unless stated otherwise the measurements where done on a Sun 3/280 under SunOS 4.0.3 (using a straightforward code generation for the Sun). Measurements are accurate within 10%.

Nfib-number	*Conditions*
90 000	The basic translation scheme as given in Section 11.1.
280 000	The integers are passed on the B-stack.
415 000	In-line code substitution is also used for the δ-rules.
2 000	Same code interpreted by the PABC interpreter/simulator (Nöcker, 1989).
300 000	The Nfib function in C (Kernighan and Ritchie, 1978).
1 200	The Nfib function in Miranda, executed by the Miranda interpreter.
6	ABC machine as in Appendix C (Miranda interpreter).
220	ABC machine as in Appendix C, with Miranda converted to the Concurrent Clean System, executed by the Concurrent Clean System.

The code for the 415 000 Nfib in ABC assembly using the extended compilation scheme becomes:

```
[           Descriptor
 "Nfib"     "a_Nfib" 1 "Nfib"       ,  || The generated descriptor
            Label                      || The apply entry
"a_Nfib",   Pop_a 1                 ,  || Reference to node containing Nfib
            Jmp "e_Nfib"            ,  || Start reducing strict argument
            Label                      || The node entry
"n_Nfib",   Set-entry "_cycle" 0    ,  || Mark node to detect cycle-in-spine
            Push_args 0 1 1         ,  || Push argument on A-stack
            Label                      || The eval args entry
"e_Nfib",   Jsr_eval                ,  || Reduce strict argument
            PushI_a 0               ,  || Copy argument to B-stack
            Pop_a 1                 ,  || Remove argument from A-stack
            Jsr "Nfib1"             ,  || Initiate calculation on B-stack
            FillI_b 0 0             ,  || Copy result from B-stack to node
            Pop_b 1                 ,  || Remove result from B-stack
            Rtn                     ,  || Rnf reached
            Label
"Nfib1",    EqI_b 0 0               ,  || Match rule 1; is argument 0?
            Jmp_false "Nfib2"       ,  || No; goto second rule alternative
            Pop_b 1                 ,  || Yes; remove argument
            PushI 1                 ,  || Store result
            Rtn                     ,  || Done
            Label
"Nfib2",    EqI_b 1 0               ,  || Match rule 2; is argument equal 1
            Jmp_false "Nfib3"       ,  || No; goto rule alternative 3
            Pop_b 1                 ,  || Yes; remove argument
            PushI 1                 ,  || Push result
            Rtn                     ,  || Done
            Label                      || Rule alternative 3
"Nfib3",    PushI 2                 ,  || Compute – n 2
            Push_b 1                ,  || Push n
            SubI                    ,  || In-line code substitution for –
            Jsr "Nfib1"             ,  || Compute Nfib (– n 2)
            Push_b 1                ,  || Compute – – n; push argument
            DecI                    ,  || In-line code substitution for –
            Jsr "Nfib1"             ,  || Compute Nfib (– – n)
            AddI                    ,  || In-line code substitution for +
            Update_b 0 1            ,  || Replace argument by result
            Pop_b 1                 ,  || Remove copied result
            IncI                    ,  || In-line code substitution for ++
            Rtn                     ]  || Done
```

With better code generation for the concrete machine the performance (including the Nfib-number) can be improved even further (see the next chapter).

Summary

- A Clean program can be translated into ABC code with the *basic translation scheme* in which Clean *graphs* are more or less directly *mapped onto graphs* in the graph store of the ABC machine.
- In this basic scheme the graph rewriting process is as follows: *redirection* is *implemented by overwriting* the root of the redex with the root of the contractum. Furthermore, *overwriting* of such a redex *is delayed* until the root normal form is reached.
- An important optimization is the use of the *B-stack* to pass arguments that are *both strict and of a basic type*. Results that are of a basic type are again returned on the B-stack. As a consequence, such basic calculations are actually not performed via graph rewriting at all but they *entirely* proceed on the B-stack.
- Many more optimizations are possible resulting in a code with an efficiency in *the same order of magnitude as classical imperative code*. Compiled code typically can be two orders of magnitude faster than when an interpreter is used.

EXERCISES

11.1* Consider the following Clean rule:

```
:: Filter INT [INT]     ->   [INT];
   Filter x  [ ]        ->   [ ];
   Filter x  [ e | es ] ->   r, IF = e x
                        ->   [ e | Filter e r ],
                             r: Filter x es ;
```

(a) Write the corresponding ABC code obeying the calling conventions.

(b) Add strictness annotations in the Clean rule where possible.

(c) Change the ABC code given above to obtain the most optimal code as explained in the textbook: use strictness annotations, in-line code substitution, B-stack parameter passing.

(d) Indicate explicitly which alterations, omissions or additions in the code are due to each optimization.