Chapter III.3
Efficiency of programs

Until now we have not been very precise on the operational model and efficiency of programs. We chosen this order since we are convinced that the basis for successful programming is the ability to write well designed programs. Nevertheless, program efficiency does matter in many situations. In this chapter we will explain the operational behaviour of Clean programs and teach you how to understand the time and space efficiency of programs. Moreover we will show you how the efficiency of programs can be improved.

In section 1 we explain graph rewriting. Graph rewriting is also called graph reduction. The basic evaluation step in a Clean program is a graph reduction according to a function in Clean. Section 2 treats the increase of the number of reduction steps as function of the value or size of the arguments of a Clean function. The basic aspects of the implementation of graph rewriting in Clean needed to understand the time and space behaviour of programs are introduced in section 3. The Clean compiler uses strictness and uniqueness information to speed up the evaluation of programs. This is explained in section 4 and 5 respectively. Finally, section 6 shows you, how programs can be transformed to increase their efficiency.

3.1 Graph rewriting in Clean

To understand the efficiency of programs as discussed in this chapter it is important to have a proper view of the operational behaviour of a Clean program. Clean is a lazy higher order functional programming language based on Functional Graph Rewriting Systems. A Functional Graph Rewriting System (*FGRS*) is a Graph Rewriting System using the functional reduction strategy [Eekelen 88, Plasmeijer 94]. A reduction strategy is a function indicating the *redex* to be rewritten. The functional strategy delays the reduction of an expression until its value is needed. However, an expression is always reduced before its value is used. The functional strategy in Clean prescribes lazy evaluation. A Graph Rewriting System (*GRS*) is an extension of Term Rewriting Systems where the terms are replaced by directed graphs in order to avoid the duplication of work via sharing of expressions [Barendregt 87a, 87b, 88]. A Term Rewriting

System (*TRS*) is a computational paradigm consisting of a collection of rewrite rules to transform terms (expressions) into equivalent terms [Klop 87].

A Clean program consists of a set of (typed) graph rewrite rules. A subgraph is a redex (*red*ucable *ex*pression) if there is a left hand side (*lhs* or *pattern*) of rewrite rule that matches this graph. A match is a mapping from the pattern to the graph that is the identity on constants and preserves the node structure. A graph is in root normal form (*rnf*) if the whole graph is not a redex and cannot become a redex by internal reduction. A graph is in normal form (*nf*) if it does not contain any redex. The rewrite rules are usually called functions.

Node-id's or node identifiers are unique references to individual nodes in the graph. Do not confuse them with the formal arguments of rewrite rules (functions). A *match* is actually a mapping from formal arguments to actual node-id's. This mapping is such that the symbols in the graph are equal to the symbols in the pattern when there is a symbol in the pattern. A graph does not change when all node-id's are changed consistently. Since the actual value of the node-id's is irrelevant the node-id's are left implicit whenever possible.

The rewrite rules are used to reduce the initial graph containing the symbol `Start` to normal form. This normal form is computed by a depth first, left to right traversal of the graph. Each node is evaluated to root normal form and its value is used as the next output element. Clean is a lazy language: the value of a graph is only computed when its value is needed. The basic goal of the Clean system is to compute the value of the `Start` rule. The functional reduction strategy is used: rewrite alternatives are tried in textual order; patterns are matched from left to right; evaluation to root normal form is forced before an actual argument is compared with non-variable part of the pattern.

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

A small example is used to illustrate graph reduction in Clean. The example is even so small that no sharing of computations occurs. The `Start` rule initiates the computation of the length of the list `[3,4]`. The function `Length` takes one argument; the list to be scanned.

```
Start :: Int
Start = Length [3,4]

Length :: [x] -> Int
Length [a:x] = 1 + Length x
Length []    = 0
```

The reduction process is illustrated by the following rewriting sequence (the redex rewritten is printed bold):

```
Start                  || a: This is the only redex, apply the Start rule
  → Length [3,4]       || b: This graph as a whole is the new redex
  → 1 + Length [4]     || c: The addition operator + forces the evaluation of its second argument
  → 1 + 1 + Length []  || d: Again the second argument of +
  → 1 + 1 + 0          || e: Second argument of second + is rewritten to (r)nf.
  → 1 + 1              || f: The graph as a whole is the new redex
  → 2                  || g: The graph is in (r)nf.
```

This rewriting process is depicted below. The graphs correspond to each of the steps shown above. The garbage that results from one rewrite step is drawn grey, it is removed in the next snapshot. The `root` node is usually not shown, but it is included here to show all redirections clearly. As a matter of fact, the only task of the `root` node is to hold the reference to the actual expression to be reduced.

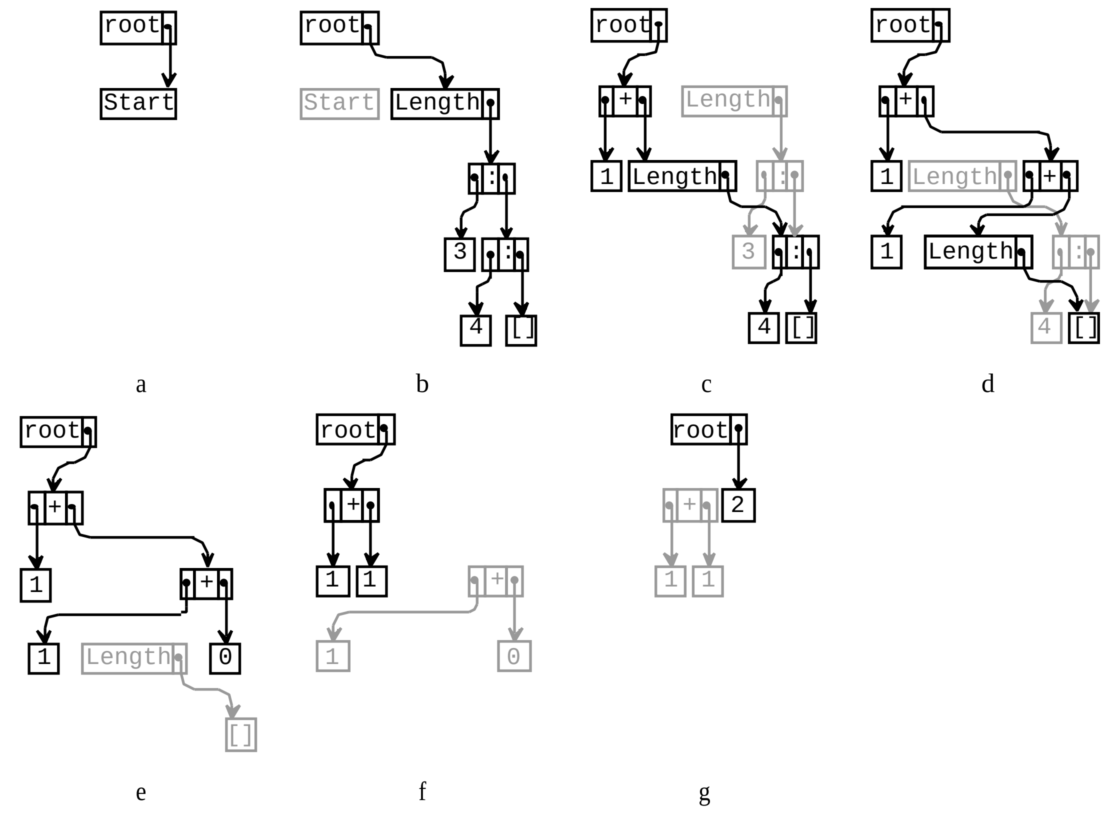


Figure III.3.1: The reduction of `Start = Length [3,4]`.
The parts of the figure correspond to the steps in the trace above.

It is important to mention that Clean is a higher order language: symbols can be used Curried; without (some of) their arguments. The operator `@` supplies one argument to a curried symbol. It is generated internally for each variable used as function and each function which more arguments then its arity. When all arguments of a function are gathered its reduction is initiated. The operator `@` is pronounced as apply. The behaviour of the apply operator is illustrated by the next example. See also the chapter about interpreting functional programs in part II.

```
twice f x    = f (f x)            ||  twice applies the function f two times to the argument x
                                  ||  The internal representation of the rhs is f @ (f @ x)
Start        = twice (I inc) 0
```

The apply operator first reduces its first argument. Next it adds its second argument as new argument to the obtained function. When the number arguments becomes equal to the arity of the function, reduction of the function is initiated. The reduction process is illustrated by the next rewrite sequence:

```
Start                              || a:  The initial graph.
 → twice (I inc) 0                 || b:  The graph after rewriting according to the Start rule.
 → f @ (f @ 0) where f = I inc     || c:  The leftmost @ reduces its first argument. .
 → f @ (f @ 0) where f = inc       || d:  The leftmost @ supplies an argument to inc.
 → inc (inc @ 0)                   || e:  The first inc needs its argument; @ adds an argument .
 → inc (inc 0)                     || f:  The argument of the leftmost inc can now be reduced.
 → inc 1                           || g:  inc 0 is evaluated; the other inc can now be evaluated.
 → 2                               || h:  The graph is in normal form
```

This reduction sequence can be depicted as shown in figure III.3.2. Apart from evaluation of Curried functions, this reduction sequence shows also some real graph reduction. The reduction of the subgraph `I inc` is shared by both `@`-operators.

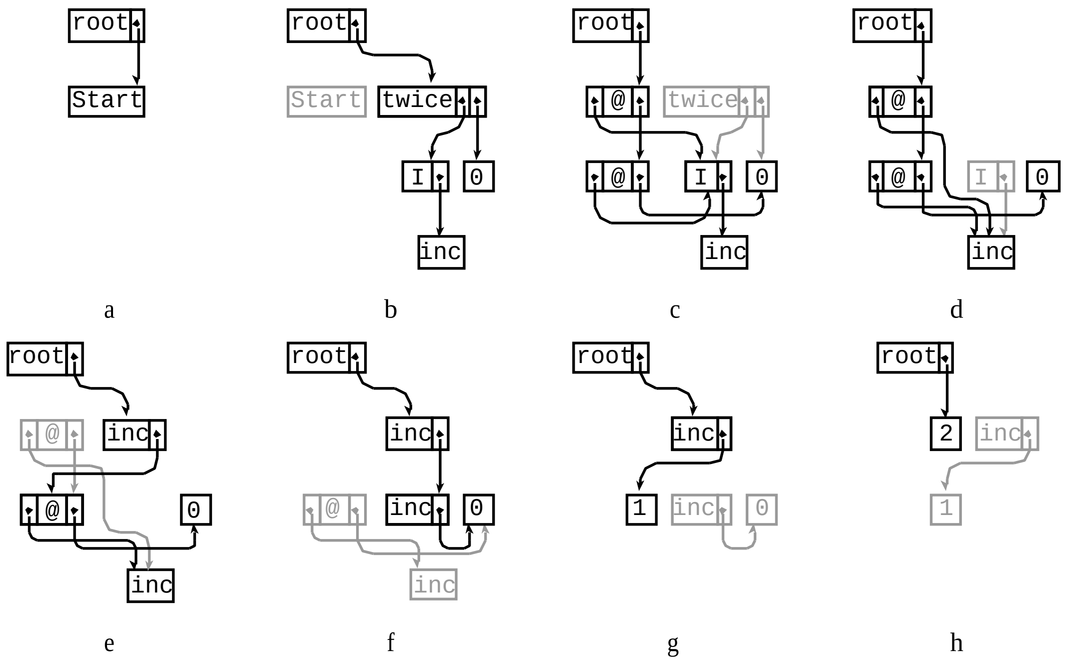


Figure III.3.2 The reduction of `twice (I inc) 0`

This example shows that it takes additional nodes in the graph and associated rewrite steps to work with Curried functions. There is nothing wrong with using Curried functions. On the contrary, Curried functions offer you a great expressive power that should be used whenever necessary. However, when you want the maximum speed for your programs you should be reluctant to use Curried function heavily. In the example above the definition of `Start` can be replaced by:

```
Start = inc (inc 0)
```

The execution of this program can be depicted as shown in figure III.3.3.

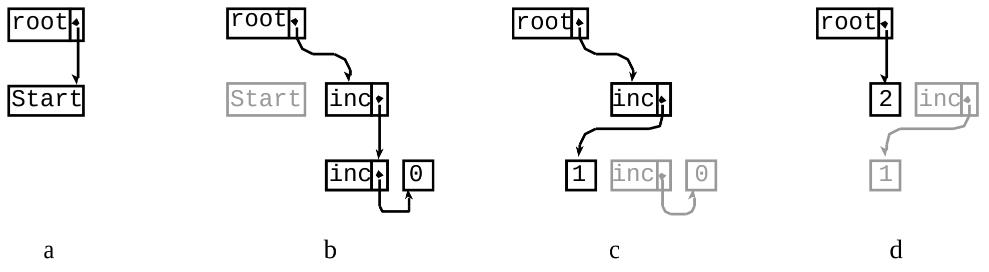


Figure III.3.3 The reduction of `inc (inc 0)`

This version of the function `Start` denotes the same value, but its is obtains more efficiently. By comparing figure 3.2. and 3.3 it is clear the second version of the function `Start` requires less nodes and less reduction steps. The presence of the identity function `I` in the reduction sequence of figure III.3.2 justifies just one additional node and one additional step. Apart from the node and reduction for `twice`, the reduction sequence depicted in figure III.3.3 contains two additional nodes for the operator `@` and two additional reduction steps for this operator.

3.2 Complexity

States a property of a given algorithm: the increase of the number of reduction steps as function of the size or value of the parameter(s). Constant factors are ignored.

In case the size of you input can grow you must use an algorithm with the lowest possible complexity to keep you program fast for large inputs. If the size of the input is fixed the complexity of the algorithm used does not matter. For a fixed size input the only thing that counts in the efficiency of the program for the given input.

The complexity is basically a property of an algorithm. However, sometimes we speak of the complexity of a problem to indicate the complexity of the best known or best possible algorithm.

Analysis

Reduction

By choosing an other algorithm.

Quick sort instead of bubble sort. (n log n instead of n^2).

Linear Fibonacci function instead of naive exponential one.

```
efib :: !Int -> Int
efib n | n<2 = 1
             = efib (n-1) + efib (n-2)

lfib :: !Int -> Int
lfib n = accfib n 1 1
  where
    accfib :: !Int !Int !Int -> Int
    accfib 0 x y = x
    accfib n x y = accfib (dec n) y (x+y)
```

3.3 Imperative reduction machine

In contrast to the previous section treating algorithmic complexity, *constant factors do matter* in real life. In this section we show you how the implementation of graph reduction in Clean rewrites graphs and how this effects efficiency. The general message to obtain efficiency is to avoid actual graph manipulations as much as possible.

The implementation of Clean uses an abstract graph reduction machine called the ABC-machine [Koopman 90, Plasmeijer 94, Koopman 95]. Clean is first compiled to code for the ABC-machine. This ABC-code is compiled to native code for your target machine. Details of this implementation are not relevant here, but some notions of this implementation are necessary to explain the efficiency of Clean programs.

The name ABC-machine is related to the three stacks used in this machine. The A-stack is the Argument-stack. The A-stack contains references to nodes involved in the current rewriting process. The B-stack contains Basic values. It is much more efficient to manipulate basic values as plain constants on a stack than to treat them boxed in nodes in the graph. The C-stack is the Control-stack. It contains return addresses in the code. In this chapter we will explain the use of the A-stack and the B-stack.

The A-stack contains references to the node involved in the current rewrite step. This stack provides an efficient way to access the nodes involved. In order to reduce some node its node-id is pushed on the A-stack. Next all function arguments are pushed on the stack. The last argument is pushed first and the first argument becomes the top of the stack. During the matching process also the subarguments are pushed on the stack.

Graph reduction on the ABC-machine differs slightly from plain graph reduction as discussed above. The differences are:

1) We use an argument stack to access nodes.

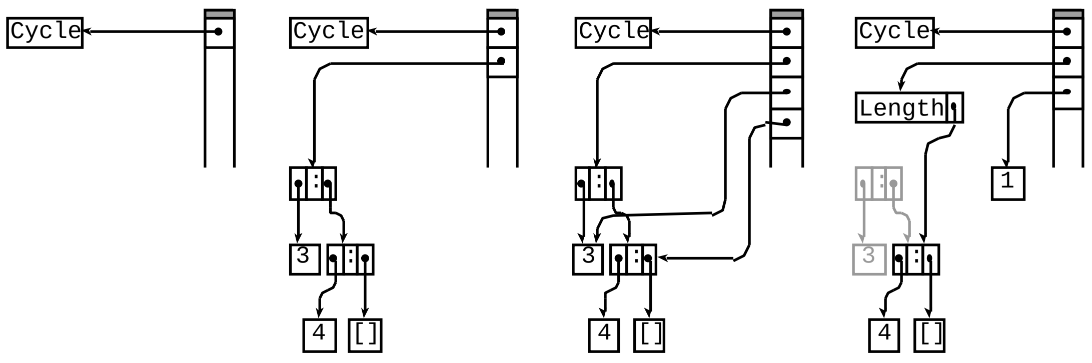
2) Nodes are updated with their reduct instead of creating a new node and redirecting all references from the redex to the newly created reduct.

3) Updating the redex is delayed until the root normal form is found. The node containing the redex is marked to detect cycles in the rewrite process. As a consequence the rule currently executed cannot be deduced from inspection of the graph, but is determined by the instruction sequence executed. For redirections, function alternatives consisting only of a variable, this cannot be done since the root of the reduct is an other node. In this situation the *root* of the reduct is copied to the root of the redex. See figure III.3.4.j and III.3.5.h for examples.

4) Garbage collection is delayed until the memory available for constructing graphs, the heap, is exhausted.

5) The dataroot is not present in the graph. A pointer on the A-stack is used instead of a node in the graph to hold the reference to the root of the graph.

As example we show again the reduction of `Length [3,4]` using the rules given above. In the figures below several snapshots of the ABC-machine state are shown during the reduction of a graph according to the code for the rewrite rule `Length`.



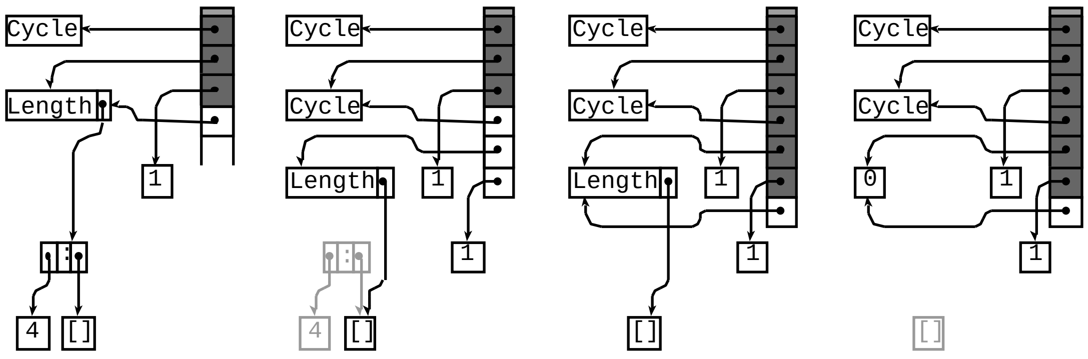
a b c d

a: The initial state. There is no need to construct a node for `root`. A pointer on the A-stack is used instead of this node. This node is marked with `Cycle` to be able to detect cyclic reduction sequences. The fact that we are actually reducing the node containing `Start` is represented by the code executed.

b: The code for `Start` has constructed the arguments of the root node (`Length`) and continues by reduction according to the code of `Length`. The root node is not updated. Updating is useless since calling the code corresponding to `Length` directly is more efficient than updating the node and inspecting its contents lateron. Moreover, after updating the node with `Length` , it must be updated again with `Cycle` in order to detect cyclic reduction properly.

c: After matching of the second alternative for `Length` also references to the sub-arguments are pushed on the A-stack. All nodes involved in the rewrite process can be accessed efficiently via the A-stack.

d: The second alternative of `Length` prepares the arguments of the operator `+` and switches to its code. Again without updating the root node.



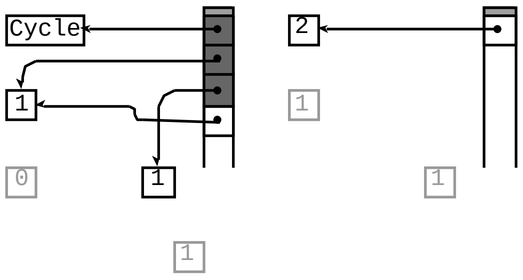
e f g h

e: The operator + needs the value of the subgraph `Length [4]`. In order to obtain this value a reference to this graph is pushed on the A-stack and its reduction is initiated. Stack frames that are currently out of scope are coloured dark grey.

f: `Length` updates the root node to detect cyclic reductions and discovers that the second alternative should by applied by inspection of its actual argument. The arguments for + are prepared and the code corresponding to + will be executed. In contrast to the real ABC-machine implementation, garbage is immediately removed.

g: This instance of + needs the value of `Length []`, and initiates its reduction after pushing a reference to this graph on the A-stack.

h: For this subgraph the first alternative of `Length` should be applied. The root node of this subgraph is replaced by its result: `0`.



i j

i: The second operator + is reduced to normal form. Its stack frame can be removed from the A-stack. The code of the first operator + can now compute the value of the subgraph and update the root node.

j: Finally, the root of the initial graph is updated with the computed normal form. Since redirection is a very expensive operation, we copy the root of the result to the root of the redex instead of the redirection. Usually copying is avoided by overwriting the root node of the redex, but for rules that consists only of a redirection this is not possible.

Figure III.3.4 Reduction of `Start = Length [3,4]` on the ABC-machine

3.4 Strictness

An argument is strict when its value always is needed in the reduction of the function. A more practical definition is: a function is strict in an argument when the reduction of the function does not terminate when the reduction of that argument does not terminate. When we denote an nonterminating expression by , a the function `f` is strict iff `f`   = . As we have seen in Part I and II, it is possible to add strictness annotations, denoted by `!`, to the type of the rewrite rules. Strictness of arguments is in general undecidable. The Clean compiler believes you when you indicate arguments as being strict.

The Clean system deviates from the basic functional strategy by reducing strict arguments eagerly to root normal form. This implies that strict arguments are reduced before the function itself is reduced. Inside the function no reduction nor inspection to see whether a strict argument is in rnf is needed. Eager evaluation of strict arguments does not change the termination properties since the value of strict arguments is needed anyway. It is operational more efficient to reduce needed arguments eagerly since it saves construction and inspection of the redex. When strict arguments of a function in a strict position have to be created, these arguments are reduced immediately. *Referential transparancy* is the property that the value of an expression does not depend on theevaluation order. Whenever you obtain a (root) normal form, it is the one and only (root) normal form of that expression. The very same feature is also called the *Church-Rosser* property. In the literature you will found some other names and proofs for this property in well behaving graph rewrite systems and λ-calculus.

The Clean compiler has its own sophisticated strictness analyzer that approximates the strictness of function arguments in a safe way [Nöcker ??, Plasmeijer 94]. A simple approximation of strictness uses the following rules:

1) Each function is strict in the first non-variable argument of the first alternative. The functional strategy needs the value of this argument to deterime whether this alternative should be applied. When the function has only variables as arguments the value of the guard will be needed (if a guard is present).

2) The root of the right-hand side occurs in a strict context, i.e. its value is needed when the value of the function application is needed.

3) The actual arguments corresponding to strict arguments of a function application in a strict context are also needed.

The strictness of arguments of a set of functions can be approximated by applying these rules until a fixed point has been reached (the strictness does not change by applying these rules). Initially all functions without strictness information are made non-strict in all their arguments. The strictness information given for existing (library) function should be used to deterimine as much strictness as possible.

The first rule determines that the function `Length` used in the examples above is strict. The second rule determines that the function `twice` is strict in its first argument and that `I` is strict in its argument. The third rule determines that `lfib` is strict in its argument since acc`fib` is strict in its first argument, due to rule 1).

Strictness can be used to optimise the reduction sequence depicted in figure III.3.3. Due to rule 1) the function length is strict in its argument. Rule 2) states that the result of the addition in the right-hand side is always needed when the result of `length` is needed. Using the strictness of the operator + also the result of the subgraph `length x` will be needed. So, it is safe to compute the value of this expression at once instead of first filling the nodes and reducing them later.

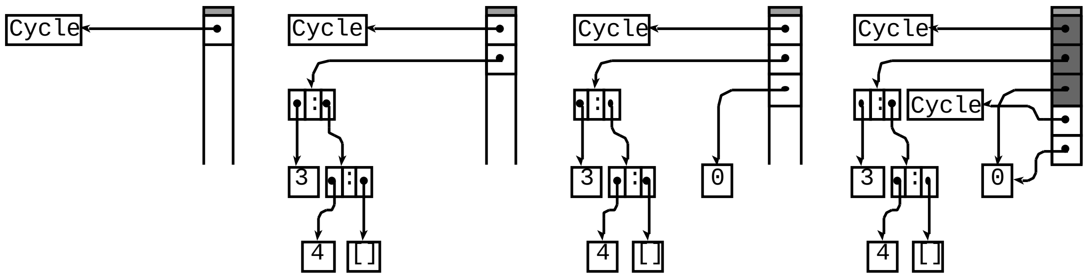
Left recursion

Figure III.3.3 shows a drawback of delaying the update of the root node until the root normal form is reached: the amount of stack space needed for the A-stack is proportional to the length of the stack. Actually, even when we would update nodes in the graph we should keep references to nodes that must be updated on the A-stack. When this is not done, we should restart looking for a redex at the root of the graph over and over again. Using an amount of stack space proportional to the length of the list can be prevented by using a left recursive version of the function length (as given in StdEnv).

```
length :: ![x] -> Int
length l = acclen 0 l

acclen !Int ![x] -> Int
acclen n []    = n
acclen n [a:x] = acclen (inc n) x
```

Reduction of the program `Start = length [3,4]` using this definition of length and eager evaluation of strict arguments is depicted in figure III.3.5.

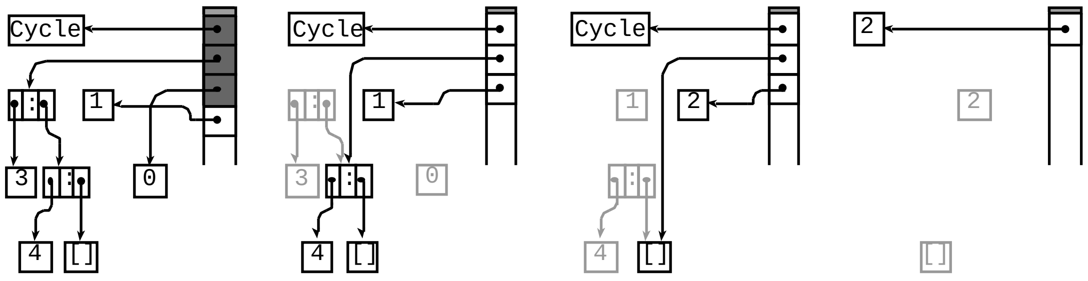


a: The initial state. A node is constructed to hold the (root of the) result of the reduction according to the function `Start`.

b: The function `Start` has created the argument of `length` and does a context switch to the code of `length`.

c: The function `length` creates the additional argument `0` for `acclen` and starts evaluation of the code of `acclen`. Since `acclen` is strict in both arguments, these arguments should be reduced to rnf before the function is called. Both actual arguments are in normal for. So, no reductions are needed.

d: The first alternative of `acclen` cannot be applied since the second argument does not match the pattern `[]`. Due to the calling conventions no reduction of the corresponding argument is needed. The function `acclen` creates the arguments for the recursive call in the second alternative. The list argument does not need any reductions. The accumulator needs to be reduced. A place holder is created for its result. A reference to it is pushed at the bottom of a new stack frame. Its argument `0` is pushed on the A-stack and reduction according to `inc` is initiated.



e: The function `inc` updates it root with the result of the reduction `inc 0`.

f: The A-stack is updates and the function `acclen` is called recursively.

g: After the reduction of `inc 1` the function `acclen` is called again. The graph to be rewritten is currently `acclen 2 []`.

h: This is reduced according to the first alternative of the function `acclen`. The root of the result is again copied to the root of the redex instead of the redirection.

Figure III.3.5 Reduction of the left recursive `length [3,4]` on the ABC-machine using strictness

Note that the reduction sequence depicted in figure III.3.5 uses an fixed amount of stack space. The stack space needed is independent of the length of the list supplied as argument. Note also that it is required to reduce the accumulator eagerly to keep the needed stack space bounded. Otherwise a subgraph of the form `1+(1+0)` is constructed and again an amount of stack space proportional to the length of the list is needed to compute the value of this graph.

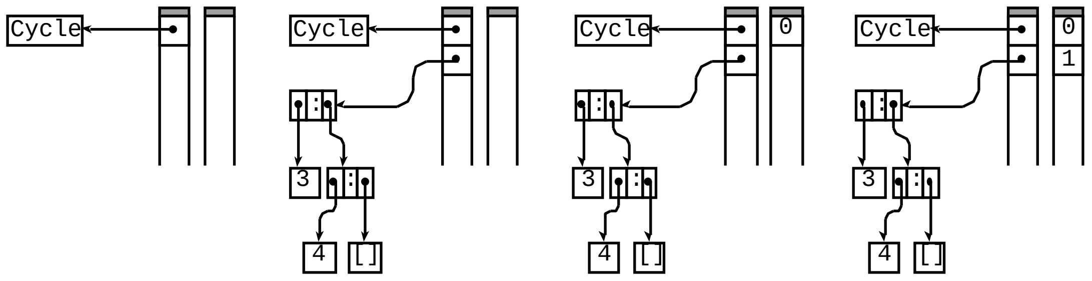
unboxed basic values

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

Basic values are manipulated always on the B-stack in the ABC-machine. 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 much data transportation. To reduce the unnecessary movement of data, the compilation scheme must be changed such that basic values stay on the B-stack as much as possible.

To achieve this, the calling conventions for the rewrite alternatives are changed: strict arguments of a basic type are passed on the B-stack and the result of a reduction is left on the B-stack when it is of a basic type. The calling conventions for the node entry and the apply entry remain unchanged. The code corresponding to the apply entry takes care of the transport of basic values between the graph and the B-stack for this function. After the reduction of a strict argument of a basic type it is transported to the B-stack. When the result of the function is of a basic type the first rule alternative entry is called as a subroutine. The rnf produced on the B-stack is transported to the node containing the redex. The complexity of the code generation is increased significantly by this new calling convention. Arguments and results must be moved to the desired place at every occurrence. This is not difficult, but involves an elaborated case analysis.

Both arguments are strict. The first argument is also a basic value. So, it will be passed on the B-stack. The result will also be passed on the B-stack. The operator `+` also expects its arguments and leaves its result on the B-stack. Only when the reduction is completely finished the result will be written in the graph since it is potentially shared.



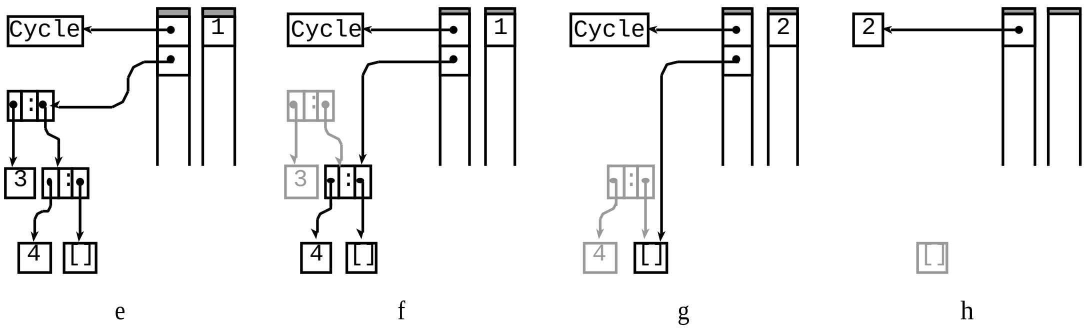
a b c d

a: The initial state. A node is constructed to hold the (root of the) result of the reduction according to the function `Start`. The only difference with figure III.3.5.a is the presence of the empty B-stack.

b: The function `Start` has created the argument of `length` and does a context switch to the code of `length`.

c: The function `length` creates the additional argument `0` for `acclen` and starts evaluation of the code of `acclen`. Since `acclen` is strict in both arguments, these arguments should be reduced to rnf before the function is called. Since the accumulator is strict and of a basic type it is passed on the B-stack instead of storing it in a node in the graph.

d: The first alternative of `acclen` cannot be applied since the second argument does not match the pattern `[]`. Due to the calling conventions no reduction of the corresponding argument is needed. The function `acclen` creates the arguments for the recursive call in the second alternative. The list argument does not need any reductions. The accumulator needs to be reduced. The value of the accumulator is computed on the B-stack instead of in the graph.



e: The function `inc` produces its result on the B-stack..

f: The A-stack and B-stack are updated and the function `acclen` is called recursively.

g: After the reduction of `inc 1` the function `acclen` is called again. The graph to be rewritten is currently `acclen 2 []`.

h: This is reduced according to the first alternative of the function `acclen`. The root is updated with the value found on the B-stack.

Figure III.3.6 Reduction of the left recursive `length [3,4]` on the ABC-machine using the B-stack.

Note that the reduction sequence in figure III.3.6 is as optimal as you can imagine. The list to process is accessed by a single pointer on the A-stack. The running length is stored in a single entry on the B-stack. The amount of stack space needed is very small, and no intermediate results are constructed in the graph.

Also the graphs indicated after the optional `let!` part of the contractum are reduced eagerly to root normal form before contractum is constructed.

3.5 uniqueness

A subgraph is unique when it is not shared by other functions. You can indicate uniqueness of function arguments by the annotation `*`. In contrast with strictness the Clean system does not trust you on your word when you indicate and argument as unique. The Clean compiler rejects programs unless it is able to verify the consistency of the uniqueness annotations.

Just like strictness, the verification algorithm for uniquness is an approximation. So, the compiler will reject some programs despite the fact that you supply the correct uniquness annotations. The reason for this difference in approach between strictness and uniquness is clear when you look at the consqquences of an erroneous annotation. An errorneous strictness annotation will force the evaluation of an expression that is not known to be used. In the worst case the value of this expression is not needed and its evaluation does not terminate. Although your program doesn't terminate while it will terminate without strictness annotations the program does not produce wrong results. For erroneous uniqueness annotations the situation is different. The Clean compiler can and will use the nodes of unique arguments in the construction of the contractum. Especially for large datastruictures like arrays this is very effective. Instead of creating a new array the array supplied as argument is updated and reused as result. It will be clear that incorrect uniqueness annotations are a disaster: you program starts producing wrong results since nodes can be updated incorrectly.

in situ update

Update an existing datastructure instead of creating a new one. Especially for large data structures like arrays.

manipulation of unique objects

3.6 Transformations to increase efficiency

General rules:

1) Think and measure before you start transforming your program. Find out where the largest part of the execution time of your program is spent. It does not make much sense to optimize the initialisation of your program a factor 10 when 98% of the runtime of your program is used elsewhere.

2) Manipulate basic values on the B-stack as much as possible (make them strict). It is even worthwhile to make functions strict in arguments of a basic type that are not always needed. Consider as example the function `accfib` above: the value of the last argument is not needed in the last recursive call. However evaluation of this argument on the B-stack is that much more efficient than treating it lazy, that it is worthwhile to do one addition, `x+y` in the last but one recursive call of `accfib`, that will not be used.

3) Avoid overloading that must be resolved at runtime. The Clean compiler will generate an appropriate warning.

4) Limit the amount of intermediate datastructures. Using the toolbox functions (like `map`, `filter`, `folr` etc.) lot of intermediate lists are created. By rearranging of the toobox functions or introduction of tailor made functions the generation of these intermediate data structures can usually be avoided. Avoiding the intermediate datastructure is more effcient since a datastructure in rnf is always constructed in the graph. Consider the introduction of continuations to remove the necessity to introduce intermediate data structures or partial solutions that must be combined.

5) Consider whether your main datastructures are the most appropriate ones. Although it is convenient to use lists to hold collections of values for program construction, accessing the elements is not very fast for long lists. Consider the use of an array or search tree instead of the list. This is one of the occasions where it pays to use abstract datastructures for the main datastructures in your program. Changing the implementation of an abstract datatype is very well localized.

6) Limit the amount of curried functions. As we have seen above using a Curried version of a function is a little more expensive than using the plain version.

7) Limit the amount of rewrite steps necessary. Each rewrite step takes some time. Even for a function that does virtually nothing stack frames have to be allocated and removed and a context switch is performed. Although these things are highly optimized and done very efficient they do take a litlle bit of time. When these things are done very often the total of these actions takes a serious amount of time. A

program can be optimized by turning functions into a macro, or by combining the the tasks of various functions into one large function. However, do not be afraid to do some additional rewrite steps to enable other optimisations. It usually is more efficient to do some testing in an additional function than repreating the test unnecessary in each call of a recursive function.

8) Develop special versions of functios that occur with fixed and predetermined arguments. Examples are some in the section fold/unfold below.

9) Think about the memory behaviour of your program. Do the subexpressions that are not needed anymore actually become garbage? We have seen some examples of this in part II. We forced the evaluation of the list representing the memory in the computer architecture and used an environment of type `[(name,value)]` instead of `name -> value` as state in the interpreter. The space consumption if of the normal form is often, but clearly not always, smaller than the space consumption of an expression equivalent to the normal form. Especially in situations where many canges of a program state are involved, it is often a good idea to force the evalaution of the state. Clear exceptions on this rule are very long, or even infinite, lists that are only partially needed.

fold/unfold

Fill in the definition of functions to obtain a more efficient tailor made function.

deforestation

Remove intermediate datastructures, especially lists.

Space behaviour is often hard to predict.