6

CLEAN — A LANGUAGE FOR FUNCTIONAL GRAPH REWRITING

T.H. Brus, M.C.J.D. van Eekelen, M.O. van Leer, M.J. Plasmeijer.

Computing Science Department, University of Nijmegen,
Toernooiveld 1, NL-6525 ED Nijmegen, The Netherlands.
E-mail: ...!mcvax!hobbit!{tom,marko,maarten,rinus}

Partially supported by the Dutch Parallel Reduction Machine Project,
sponsored by the Dutch ministry of Science and Education.

Abstract.

Clean is an experimental language for specifying functional computations in terms of graph rewriting. It is based on an extension of Term Rewriting Systems (TRS) in which the terms are replaced by graphs. Such a Graph Rewriting System (GRS) consists of a, possibly cyclic, directed graph, called the data graph and graph rewrite rules which specify how this data graph may be rewritten. Clean is designed to provide a firm base for functional programming. In particular, Clean is suitable as an intermediate language between functional languages and (parallel) target machine architectures. A sequential implementation of Clean on a conventional machine is described and its performance is compared with other systems. The results show that Clean can be efficiently implemented.

6.1 INTRODUCTION

In order to be able to reason about (future) functional languages and their implementations as well as for the comparison of new machine architectures (reduction machines), it is necessary to choose a computational model. Functional languages and their implementations have very little in common with the familiar Turing machine model of computation. The λ-calculus is often seen as the computational model for these languages (Peyton Jones (1987a)). However, most implementations are not really based on λ-calculus but on combinatory logic (Turner (1979a), Johnson (1984), Cousineau *et al.* (1985)). Furthermore graphs are used for the representation of functional programs in which redundant computations are prevented via sharing of subgraphs. The presence of patterns in functional languages is very essential. Though it is possible to translate them to ordinary tests it appears to be worth-while to incorporate patterns in the computational model. Consequently, if one wants to have a computational model for functional languages which is also close to their implementations, pure λ-calculus is not the obvious choice anymore.

Another reason for reconsidering the computational model is that functional languages are still being further developed. Several researchers investigate how to incorporate concepts such as parallelism and unification (Hudak & Smith (1986), de Groot & Lindstrom (1986)). These appreciated concepts in some declarative languages are not straightforward to incorporate in functional languages nor in the underlying computational model of the λ-calculus.

Hence, we have developed an alternative computational model by extending Term Rewriting Systems (O'Donnell (1985), Klop (1985)) to a model of general graph rewriting. Via this general model it must be possible to reason about differences between languages, to prove correctness, to port declarative programs to different (parallel) machines. Lean (the Language of East-Anglia and Nijmegen) (Barendregt *et al.* (1987b)) is a first proposal for a language based on such a model. It is the result of collaboration between two research groups: the Declarative Alvey Compiler Target Language group at the University of East-Anglia (Glauert *et al.* (1985)) and the Dutch Parallel Reduction Machine group at Nijmegen.

The language Clean presented in this paper is roughly the subset of Lean intended for functional languages only. In Clean, graph representations of terms are used to perform term rewriting more efficiently. The design of Clean, done in parallel with the Lean language, was triggered by the need for an intermediate language and corresponding computational model in the Dutch Parallel Reduction Machine Project. This project, a collaboration between the Dutch Universities of Amsterdam, Utrecht and Nijmegen, has as its goal the development of a parallel reduction machine. An overview of the results of the project is given in Barendregt *et al.* (1987c).

The basis of Clean is that a computation is represented by an initial data graph and a set of rules used to rewrite this graph to its result. The rules contain graph patterns that may match some part of the graph. If the data graph matches a rule it can be rewritten according to the specification in that rule. This specification makes it possible to first construct an additional graph structure and then link it into the data graph by redirecting arcs from the original graph. Clean describes functional graph rewriting in which only the root of the subgraph matching a pattern may be overwritten. The semantics allow parallel rewriting where candidate rewrites do not interfere. The rewriting process stops if none of the patterns in the rules match any part of the graph which means that the graph is in normal form.

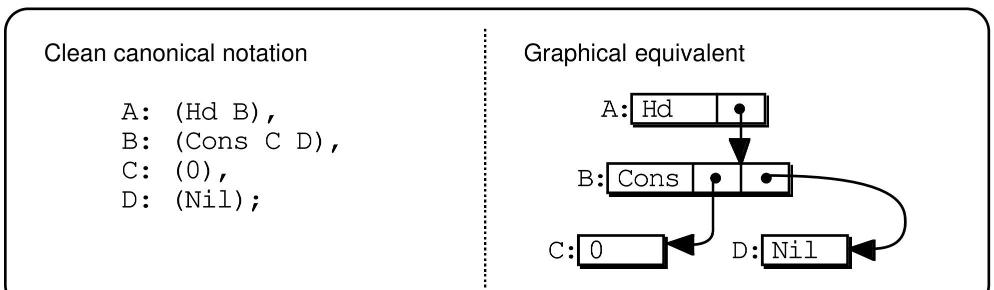
In this paper we first informally introduce the language Clean giving some examples how graph rewriting is performed. The general semantics of the graph rewriting process is explained in Barendregt *et al.* (1987b). A formal description of the basis and theoretical properties of the graph rewriting model followed in this paper can be found in Barendregt *et al.* (1987a). After the introduction to the language some examples are given to show its expressive power. Hereafter an implementation of Clean on a conventional machine is discussed. Its speed will be compared to other implementations of functional languages.

6.2 GENERAL IDEA OF THE LANGUAGE

6.2.1 CLEAN GRAPHS

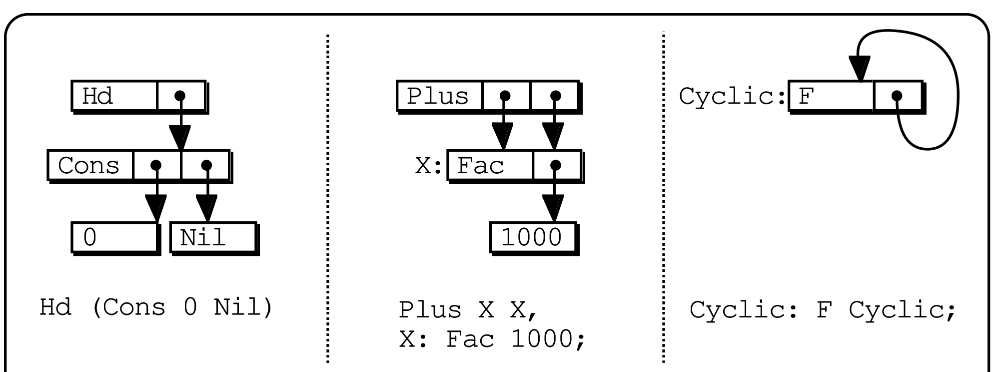
The object that is manipulated in Clean is a connected, possibly cyclic, directed graph called the *data graph*. When there is no confusion, the *data graph* is simply called *the graph*. Each node in the graph has an unique identifier associated with it (the *node identifier* or *nodeid*). Each node contains a *symbol* and a possibly empty sequence of nodeid's (the *arguments* of the symbol) which define directed arcs to nodes in the graph. Symbols have fixed arities. The data graph is a *closed graph* i.e. contains no variables, this in contrast with the Clean graphs specified in rules.

Programming with pictures is rather inconvenient so we have chosen for a linear notation for graphs. In the most extensive form of this notation (the canonical form) graphs are represented by giving the list of the nodes out of which the graph is built.



graph example

In order to get a more readable form we may substitute the contents of a node for a nodeid mentioned in a node and furthermore we only explicitly have to notate nodeid's if we need them to express sharing. Brackets are left out if they are redundant. This way of representing graphs has the advantage that it is very comprehensive. Note that each Clean graph described in this way can be transformed to an equivalent graph notated in Clean's canonical form. The syntax of Clean is given in appendix A.



examples of Clean graphs

6.2.2 CLEAN PROGRAMS

Although for the understanding of the rewriting process it is important to know what a data graph looks like, the data graph itself is never specified in a Clean program. The initial data graph is a given object generated by the operating system as we will explain later. Consequently a *Clean program* only consists of a set of *rewrite rules*. Each rewrite rule specifies a possible transformation of the data graph.

```
Hd (Cons a b)       -> a                                             ;

Add Zero n          -> n                                             |
Add (Succ m) n      -> Succ (Add m n)                                ;

Fac 0               -> 1                                             |
Fac n               -> *I n (Fac (-I n 1))                           ;

F (F x)             -> x                                             ;

Start stdin         -> Add (Succ Zero) (Succ (Succ Zero))            ;
```

The left-hand-side of a rewrite rule consists of a Clean graph which is called a *redex pattern*. The right-hand-side either consists of a Clean graph called *contractum pattern* or the right-hand-side contains only a *redirection*. The patterns are said to be *open* since they contain variable nodeid's expressed by the identifiers starting with a lower-case letter. A redirection is not a graph but just consists of a single nodeid variable. The first symbol in a redex pattern is called the *function symbol*. Rules starting with the same function symbol are collected together forming a *rule-group*. The members of a rule-group are separated by a '|'. Symbols other than function symbols are called *constructors* because they are usually used to construct data structures or data types. Note that function symbols may also occur at other positions than the head of the pattern. At such occurrences function symbols are also called *constructors*. The use of the start rule and its special argument is explained in the section on input/output.

6.2.3 REWRITING THE DATA GRAPH

The initial graph of a Clean program is rewritten to a final form by a sequence of applications of individual rewrite rules. For a rule to be included in the sequence, there must be a correspondence between a redex pattern of the rule and some subgraph of the data graph.

An *instance* of a redex pattern is a subgraph of the data graph for which there exists a mapping from the pattern to that subgraph in such a way that the mapping preserves the node structure (corresponding nodes must have the same arity) and that it is the identity on constants. This mapping is also called a *match*. The subgraph which matches a redex pattern is called a *redex* (*red*ucable *ex*pression) for the rule concerned.

Assume that we have the following Clean rules:

```
Add Zero n          -> n                                                    |     (1)
Add (Succ m) n      -> Succ (Add m n)                                       ;     (2)
```

and assume that we have the following data graph

```
Add (Succ Zero) (Add (Succ (Succ Zero)) Zero)
```

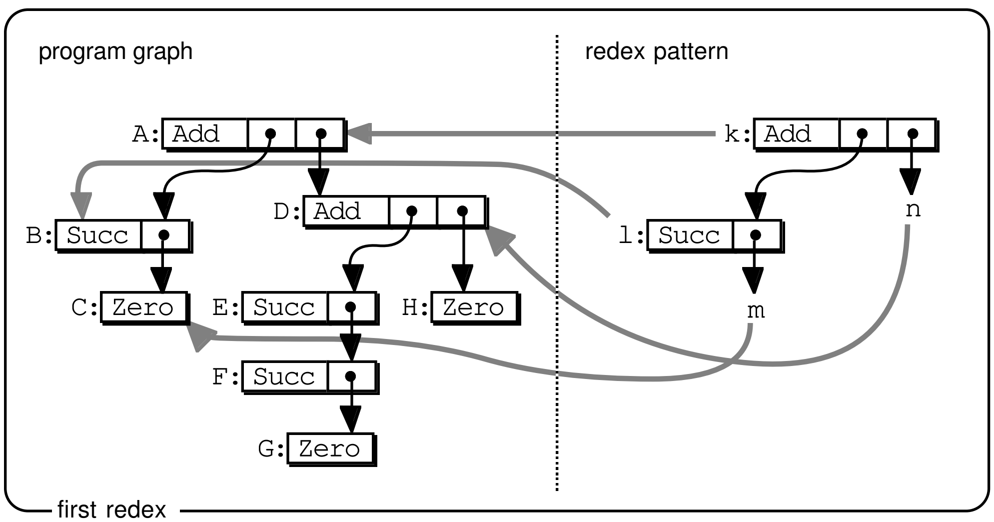
There are two redexes, both matching rule 2:

```
Add (Succ __m_) ____________n__________________
Add (Succ Zero) (Add (Succ (Succ Zero)) Zero)
```

and:

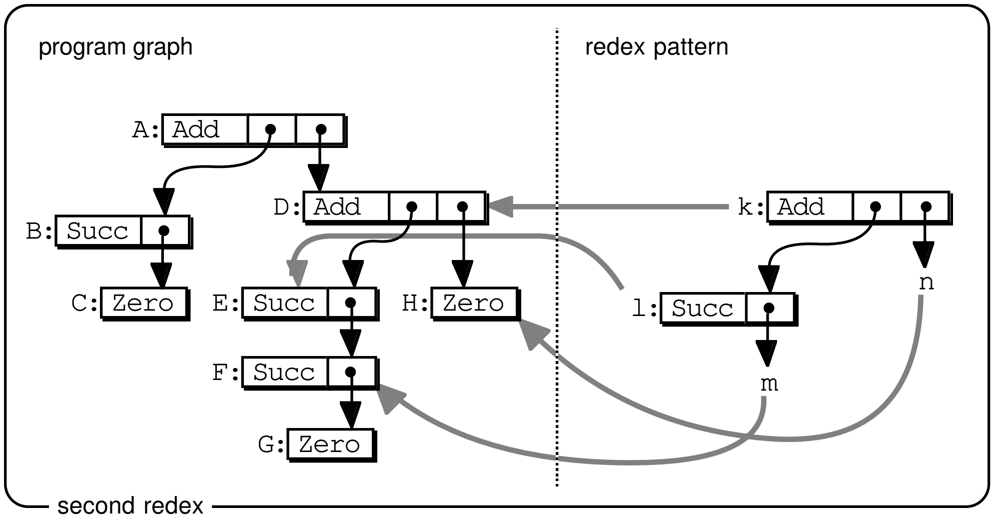
```
                  Add (Succ ____m______) __n__
Add (Succ Zero) ( Add (Succ (Succ Zero)) Zero)
```

In graphical form the first redex can be found by performing the following mapping:



We see that the redex pattern of rule 2 matches the indicated subgraph of the data graph if we substitute the following nodeid's of the graph for the variable nodeid's in the redex pattern: `k` := `A`, `l` := `B`, `m` := `C` and `n` := `D`. Note that in order to perform this mapping we have to use the canonical form of the graphs. This means for nodeid's not explicitly mentioned in the patterns new unique variable nodeid's (in the example `k` and `l`) have to be invented.

The redex pattern of rule 2 can also be mapped on another part of the data graph if we substitute `k` := `D`, `l` := `E`, `m` := `F` and `n` := `H`, as shown in the next picture.

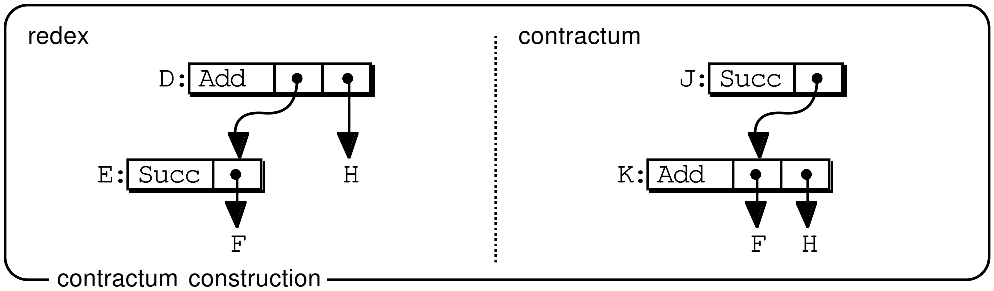


If a particular rule is applied to a matching redex, the graph is rewritten according to the right-hand-side of that rule. If this right-hand-side consists of a *contractum* pattern, the first step is to create an instantiation of this pattern which is called the contractum. The contractum is a new Clean graph as specified in the right-hand-side in which the nodeid variables defined on the left-hand-side are replaced by the corresponding matching nodeid's in the redex. New nodeid constants are created for those nodeid variables in the right-hand-side which are not defined in the left-hand-side.

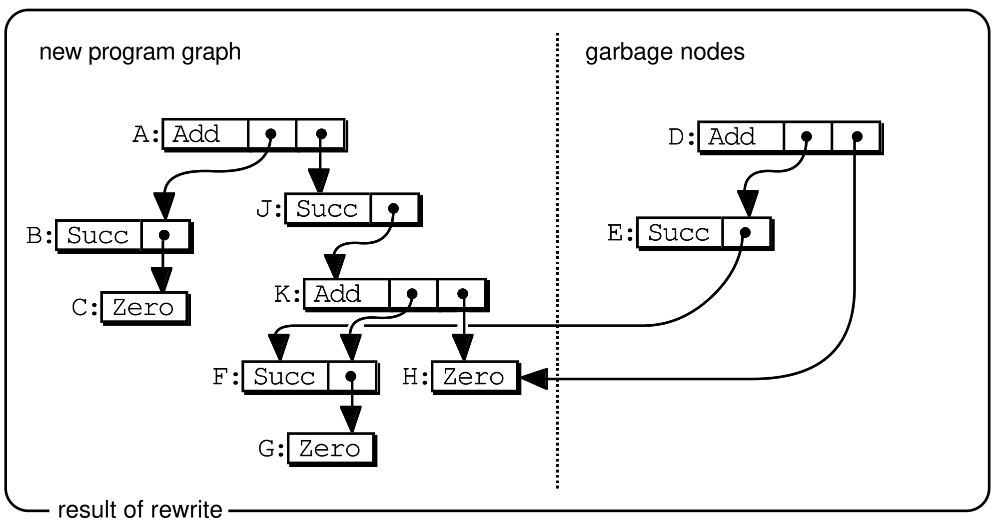
The new data graph is finally constructed by taking all arcs pointing to the root node of the redex and redirecting them to the root node of the contractum. This has the effect of "overwriting" the root of the redex with the root of the contractum. If the right-hand-side is a redirection no contractum has to be built. All arcs pointing to the root node of the redex are now redirected to the single nodeid that matches that nodeid variable. This "overwrites" the root of the redex with the root of a subgraph of the data graph. This concept of redirecting has the advantage over the usual "overwriting of node's" semantics that we do not have to deal with indirection nodes on the semantic level.

After the rewriting, nodes which are no longer reachable from the root of the data graph are considered to be garbage and may be collected by a garbage collector.

We see that in the example above the second redex matches the data graph if we take the following mapping from the nodeid variables to the nodeid of the data graph: `k` := `D`, `l` := `E`, `m` := `F` and `n` := `H`. The right-hand-side of the second rule specifies that in this case the contractum can be constructed as follows:



For the variables `m` and `n` in the right-hand-side we have to take the same mapping (`m` := `F`, `n` := `H`). For the other variables (say `o` and `p`, they are not specified explicitly) we invent new unique nodeid's (say `J` and `K`). Now the contractum is glued to the data graph by redirecting all nodes pointing to the root of the redex (`D`) to the root of the contractum (`J`). All nodes not reachable from the root of the data graph are considered to be garbage. If we remove these nodes (`D` and `E`) we finally have the new data graph and can start another rewriting.



The graph after rewriting is called the *result*. The process of performing a rewriting is often called a *reduction* step. A data graph containing no redexes is said to be in *normal form*. The rewriting process will start with the start rule and rewriting is performed repeatedly until the strategy has transformed the data graph to normal form.

6.2.4 REDUCTION STRATEGIES

In general there will be several possible redexes in the graph. It may even be the case that one and the same redex can be reduced according to more than one rule; a typical situation which is called an ambiguity in the literature. An algorithm which repeatedly rewrites the graph making choices out of the available redexes and out of all the possible matches of those redexes is called a *rewriting strategy* or a *reduction strategy*. Note that this definition of strategy is somewhat more liberal than some definitions circulating in the literature. It allows the strategy to choose out of several possible matches of one and the same redex. Furthermore, it is also not necessary for a strategy to rewrite the graph until a normal form is reached, which e.g. allows strategies that reduce to head normal form only.

Given a set of rules (including a start rule), an initial graph and a rewriting strategy we have a system with a dynamic behavior, a *rewriter*. Although it is sometimes only implicitly defined, every implementation of a rewriting system must rewrite according to a given strategy. If the strategy is deterministic, every program (including a so-called ambiguous one) will always have exactly the same result.

Every Clean program is reduced with one and the same strategy. This strategy is called the *functional* strategy, because it resembles very much the way in which normally reducing is performed in lazy functional languages. Below we will give an operational definition of the functional strategy. A formal description can be found in Goos & van Latum (1987) using a formal method described in van Eekelen & Plasmeijer (1986).

The functional strategy proceeds as follows: the strategy considers one or more candidates for rewriting. When a match is found rewriting is performed as described in the previous section. The functional strategy starts with reducing the root node of the graph to head normal form (RtoHNF). The result will be a graph with the property that its root is not part of any redex. Thereafter this reduction to head normal form is recursively called on the arguments of the obtained result (from left to right).

The RtoHNF starts with the examination of the graph it is applied to: if the symbol in the root node of that graph is a constructor the reduction is finished. If the symbol is a function symbol the corresponding rewrite rules for that function are examined in order to see if the given graph is a redex or can become a redex. In *textual order* the corresponding rules are examined to see if one of the redex patterns matches or can be made to match. The graph is rewritten according to the first rule that matches and hereafter the RtoHNF is recursively applied to the subgraph with the redirected nodeid as root. If no rule can be made to match the reduction is finished.

In order to examine the matching of redex pattern and graph the redex pattern is traversed in preorder and, possibly after forcing evaluation of corresponding parts in the graph, redex pattern and graph are compared. If there is a variable in the pattern, the traversal is continued. If a function symbol is encountered in the graph where there is a symbol in the pattern, the RtoHNF recursively calls itself to force evaluation of this function. This aspect of the functional strategy is remarkable because evaluation is forced during a matching attempt. The resulting graph will be in head normal form. Hereafter a symbol encountered in the pattern must be the the same symbol as in the graph. If they are different a match is impossible and the next rule is tried. If they are the same the traversal is continued. If the traversal reaches the end of the pattern a match is found. The result of this lazy evaluation scheme is that after the traversal we might end up with a redex after all and the rule can be applied.

In the following example a data graph is constructed in which parts are shared. Note that when the data graph is actually a tree there is no difference with a term rewriting system.

```
Start stdin        -> Double (Add (Succ Zero) Zero)          ;    (A)
Double a           -> Add a a                                ;    (B)
Add Zero n         -> n                                      |    (1)
Add (Succ m) n     -> Succ (Add m n)                         ;    (2)
```

Rewriting a shared part will reduce the number of rewriting steps compared to an equivalent term rewriting system. The rewriting will take place as specified below. Note that when a nodeid variable appears more than once at a right-hand-side, the rewriting process will generate a contractum in which the corresponding matching node is shared.

```
Start Nil                                                    →    (A)
Double (Add (Succ Zero) Zero)                                →    (B)
Add X X, X:Add (Succ Zero) Zero                              →    (2)
Add X X, X:Succ (Add Zero Zero)                              →    (2)
Succ (Add M X), X:Succ M, M:Add Zero Zero                    →    (1)
Succ (Add Z X), X:Succ Z, Z:Zero                             →    (1)
Succ (Succ Zero)
```

Although this functional strategy will look very familiar for people acquainted with functional languages, it really is a very peculiar strategy in the TRS and GRS world. To have a priority in the rewrite rules leads in general to a rewrite system without proper semantics (Klop (1985)). In this case the system is sound due to the forced evaluation of the arguments of a function as described above. Although we theoretically prefer a TRS without such a priority in rules, we have adopted the functional strategy because it is used so often in practice.

6.2.5 DATA TYPES

Constructors are not only handy to create datastructures in the form of directed, possibly cyclic graphs, such as list and tuples, but they can also be used to represent any other object or to indicate the type of an object. For instance, one can define numbers as:

```
0   -> Num Zero                                              ;
1   -> Num (Succ Zero)                                       ;
2   -> Num (Succ (Succ Zero)                                 ;
...
```

Here the constructor `Num` (also called a *type constructor*) indicates the type of the number objects while the constructors `Succ` and `Zero` (also called *data constructors*) are used to represent numerical values. A function for doing addition that yields a result of type `Num` could look like:

```
Add      (Num x)       (Num y)   -> Num (Add2 x y)           ;
Add2     Zero          y         -> y                        |
Add2     (Succ x)      y         -> Succ (Add2 x y)          ;
```

In Clean one is not obliged to specify the arguments of a constructor in a redex pattern if they are not used elsewhere in the rule. This is in particular a handy notation when one wants to write rules for objects of a certain type. For example instead of:

```
Fac 0              -> 1                                           |
Fac n:(Num x)      -> Times n (Fac (Minus n 1))                   ;
```

one may write:

```
Fac 0              -> 1                                           |
Fac n:Num          -> Times n (Fac (Minus n 1))                   ;
```

The value can be passed to a function by passing the corresponding nodeid (`n` in the example). Note that in this example the type of the argument is checked at run-time in the matching phase. Of course this check can be prevented by not using the `Num` constructor in the pattern or the objects.

6.2.6 BASIC TYPES AND PREDEFINED DELTA RULES

For practical reasons it is convenient that rules for performing arithmetic on primitive types (numbers, characters etc.) are predefined such that they can be implemented efficiently, preferably by using the integer and real representation and corresponding arithmetic available on the computer.

In Clean for primitive types a number of constructors such as `INT`, `REAL` and `CHAR` are predefined with hidden arity. Objects of these primitive types can be denotated: for instance `5` (an integer), `5.0` (a real), `'5'` (a character). The standard basic functions for arithmetic defined on these basic types are also predefined. These predefined rules are called *delta rules* .

The possibility in Clean to leave out the specification of the arguments of a constructor in a redex pattern is mandatory for primitive type constructors. As a consequence how an object of a certain primitive type is represented will be hidden for the Clean programmer. Besides this special restriction, added only for software engineering reasons, primitive type constructors act as ordinary constructors.

6.2.7 INPUT AND OUTPUT

Input and output is always somewhat problematic in functional languages. We have chosen for a solution in which the operating system builds the initial graph. The initial graph contains the standard input as shown below.

```
Root:    Start Stdin,
Stdin:   Cons "line1\n" (Cons "line2\n" (Cons ………));
```

The input can be accessed in the Clean program via the argument of the `Start` rule. The output generated by a Clean program is in principle a depth-first representation of the normal form to which the initial data graph is reduced. As soon as the initial graph is in head normal form the head symbol is printed and hereafter the printing process is recursively applied to the arguments of that symbol. In the near future it will be possible to associate printing actions with predefined constructors like in Miranda (Turner (1985)).

6.2.8 ANNOTATIONS

In Clean to every node an attribute can be assigned via an annotation. Annotations have in general the form of a list of strings between curly braces. Annotations are to be considered as compiler and run-time directives (pragmats). The number and type of annotations are left open and will depend on the actual implementation. Although annotations may influence the efficiency and strategy of the rewriting process, they are of course not allowed to influence the outcome of a computation. It is all right for a Clean compiler to ignore annotations.

At this moment in our compiler only one annotation is implemented indicating that the annotated argument is needed for the computation (`"!"` or `"{strict}"`). Future annotations are planned for work to be done in parallel, for load distribution, etc.

6.3 EXAMPLES OF CLEAN PROGRAMS

6.3.1 MERGING LISTS

The following Clean rules are capable of merging two ordered lists of integers (without duplicate elements) into a single ordered list (again without duplicate elements)* :

```
Merge Nil          Nil          -> Nil                               |
Merge f:Cons       Nil          -> f                                 |
Merge Nil          s:Cons       -> s                                 |
Merge f:(Cons a b) s:(Cons c d) -> IF  (<I a c)
                                       (Cons a (Merge b s))
                                       (IF (=I a c)
                                           (Merge f d)
                                           (Cons c (Merge f d)))   ;
```

Note that in the last rule the arguments as a whole as well as their decomposition is used.

6.3.2 HIGHER ORDER FUNCTIONS, CURRYING

In this example we show how higher-order functions are treated in Clean, by giving the familiar definition of the function map.

```
Map f       Nil        -> Nil                                    |    (1)
Map f       (Cons a b) -> Cons (Ap f a) (Map f b)                ;    (2)
Ap  (*IC a) b          -> *I a b                                 ;    (3)
Start stdin            -> Map (*IC 2) (Cons 3 (Cons 4 Nil))      ;    (4)
```

This will be rewritten in the following way:

```
Start Nil Nil Nil                                          →    (4)
Map (*IC 2) (Cons 3 (Cons 4 Nil))                          →    (2)
Cons (Ap L 3) (Map L (Cons 4 Nil)), L:(*IC 2)              →    (3)
Cons (*I 2 3) (Map L (Cons 4 Nil)), L:(*IC 2)              →    *I
Cons 6 (Map L (Cons 4 Nil)), L:(*IC 2)                     →    (2)
```

---

* `<I` and `=I` are delta rules for integer comparison, `IF` is a delta rule for the conditional.

```
Cons 6 (Cons (Ap L 4) (Map L Nil)), L:(*IC 2)                    →   (3)
Cons 6 (Cons (*I 2 4) (Map L Nil)), L:(*IC 2)                    →   *I
Cons 6 (Cons 8 (Map L Nil)), L:(*IC 2)                           →   (1)
Cons 6 (Cons 8 Nil)
```

`*I` is a predefined delta rule which multiplies two integers. Rule 3 of this example will rewrite `(Ap (*IC 2) 3)` using the constructor `*IC` which is the curried version of `*I`, to its uncurried form `(*I 2 3)` making the multiplication possible. One will need such an "uncurry" rule for every function which is used on a curried manner. Note that during rewriting the node `L:(*IC 2)` is shared. In this case sharing only saves space, but not computation.

6.3.3 GRAPHS WITH CYCLES

The following example is is a solution for the Hamming problem: it computes an ordered list of all numbers of the form 2^n3^m, with $n,m \geq 0$. We use the map and merge functions of the previous examples.

```
Ham -> Cons 1 (Merge (Map (*IC 2) Ham) (Map (*IC 3) Ham))        ;
```

A more efficient solution to this problem can be obtained by creating a cycle in the contractum. With these cycles we make heavy use of computations already performed. The new definition is:

```
Ham -> x: Cons 1 (Merge (Map (*IC 2) x) (Map (*IC 3) x))         ;
```

6.3.4 COMBINATORY LOGIC

Finally we show the Clean equivalent of a well-known TRS.

```
Ap (Ap (Ap S a) b) c   -> Ap (Ap a c) (Ap b c)                   |
Ap (Ap K a) b          -> a                                      ;
```

6.4 THE IMPLEMENTATION OF CLEAN

This section will describe the current implementation of Clean. This implementation was developed as a testbed for the definition of Clean. It was partly constructed concurrently with the language itself. The advantage was that the definition of Clean could often be corrected or adjusted when an inconsistency was overlooked and became apparent in the implementation.

The Clean compiler was developed on a VAX/750 running UNIX BSD 4.2. UNIX and VAX specific aspects will now and then surface in the implementation and in the following sections. We have tried to minimize this.

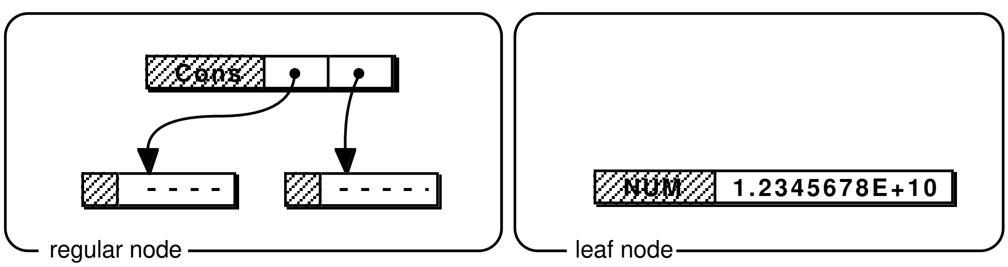
6.4.1 CLEAN RUN-TIME PHILOSOPHY

Clean is a graph rewriting language, therefore in principle we need a heap to build graphs in. The initial graph is built by the run-time system. Under control of the reduction strategy this graph will be transformed to normal form. These transformations are performed by the compiled code, using a heap and 2 stacks (a system stack and an argument stack). The functional strategy is compiled into this code. This means that for the implementation of a new strategy it is necessary to change the compiler or at least its code generator.

The basic implementation algorithm looks for a matching redex according to the functional strategy. It will overwrite the matching redex with the corresponding right-hand-side, thereby realizing redirection. This continues until there is no redex left. The main work that is being done this way is building graphs. Hence the code will not be fast, because the system is continuously allocating nodes in the heap. As a stack mechanism is inherently faster then a heap mechanism, at least in Von Neumann like machine architectures, we have tried to put the graph on a stack instead of in a heap whenever possible. The main issue in this respect is the LIFO access characteristic of a stack opposed to the random access in a heap. We had to find LIFO behaving mechanisms in our language, or its implementation. Lazy evaluation does not behave LIFO, eager evaluation does. This is the reason we need a strictness analyzer, which could free us from a lot of laziness, and give us eagerness instead. In 4.5 we discuss how we used strictness.

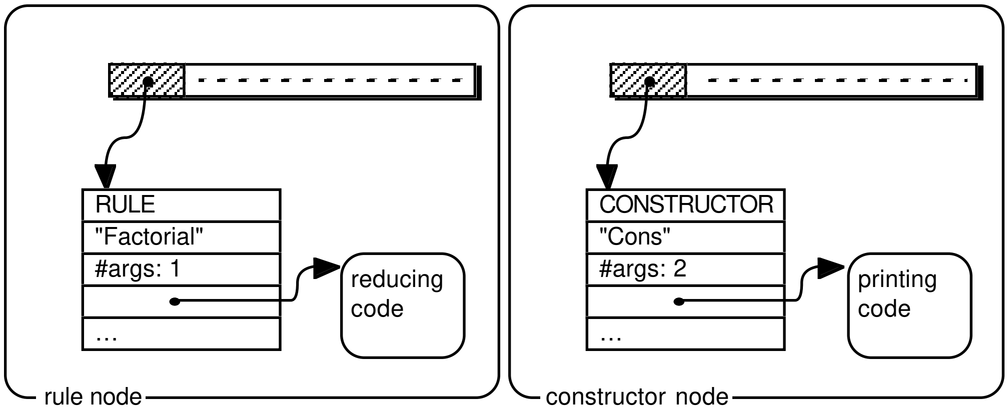
6.4.2 GRAPHS

In a Clean graph we can distinguish regular nodes and leaf nodes. Every node has a symbol field which indicates the kind of symbol stored in the node. It is implemented as a pointer to a record containing all the necessary symbol information. If the symbol field labels a node as a regular node it can only be filled with references to nodes. If the symbol field indicates a leaf node then the rest of the node has no node reference at all. The other bits of the node will contain information like a number or a character code.



This strict division is made to enable the garbage collector to easily and quickly follow all the necessary links in the heap.

Regular nodes are either rule or constructor instantiations. Rules have code associated with them which needs to be called for reduction to head normal form. Constructors have code, that will be called when the constructor needs to be printed. This includes code to evaluate arguments.



Graphs are built from right to left, from bottom to top using the argument stack. This works fine if we have no sharing and cycles in the right-hand-side. If a certain subtree is shared, we will save a reference to this subtree as soon as it is built. If the subtree is needed again, the saved reference can be taken. Cycles can be solved by inserting a place holder on the argument stack whenever we find a link back to a former node. We save a reference to the node with the place holder in it. As soon as the node to which the link back referred has been built, the place holder is replaced by the actual reference.

6.4.3 REDUCING GRAPHS

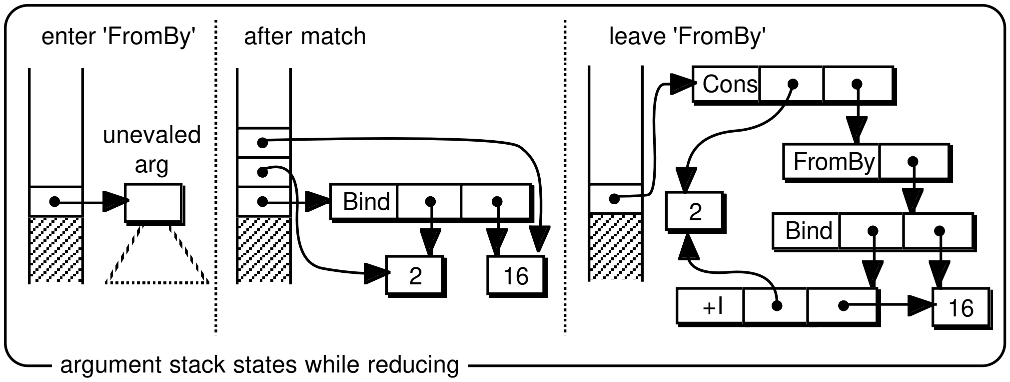
Reducing first involves finding a redex, using the functional strategy, by matching the formal and actual arguments of a rule. Every formal argument is a graph of node patterns. A node pattern can either be a variable or a pattern. In case of a variable the reference to the actual argument is copied to the argument stack. In case of a pattern, the actual argument is first reduced to head normal form. If the result matches the pattern a pointer to the actual argument is copied to the argument stack.

After a match has been found the rule must be rewritten. Due to the match the argument stack contains references to all the left-hand-side variables. The rewrite code of the rule will use these during its rewrite. Having rewritten the rule all the references are popped of the stack and the result is pushed on top of it. The following picture will illustrate this* :

---

* `+I` is the delta rule for integer addition.

```
FromBy (Bind f b) -> Cons f (FromBy (Bind (+I f b) b));
```



The above scheme works fine for eager evaluation. We actually have the top node available on the stack at all times. Using lazy evaluation we sometimes have to rewrite a node which has already been built, therefore we have to adapt this scheme. First the contents of the node in the graph is copied to the stack, then it is rewritten. This will return a new node, in head normal form, on the stack. But the real top node is still untouched in the heap. The redirection is implemented by making the old node an indirection node pointing to the new node. Overwriting the old node is in general impossible because the new node could be bigger than the old one.

6.4.4 Heap management

The heap delivers variable sized nodes. Once created, a nodes size can not be changed. Heap management routines take care of garbage collection in the heap. The garbage collector is based on a simple mark/scan algorithm.

The memory management used is an ad hoc solution, which happens to perform satisfactory. It could be streamlined significantly, or even be replaced altogether, to get a better performance. A fast memory management is essential.

Here it becomes clear why we can not merge the argument stack with the system stack, why we need a separate stack with node references. Our compilation scheme does not guarantee that all non-garbage nodes can be found from the root of the data graph. Therefore the garbage collector will have to look in the stack for references to find all non-garbage nodes. Because it is impossible for the garbage collector to identify items on a stack as node references or other values, such as reals or integers, we save references to nodes on a special stack.

6.4.5 Optimisations using strictness

As we have seen, we want to make use of the stack, and ban the use of the heap, as much as possible. Lazy evaluation prohibits this, eager evaluation enables this. This led us to methods of trading laziness for eagerness where ever possible, without endangering the termination of the reduction process.

The functional strategy enables us to compile the right-hand-side of rules in an efficient way. To illustrate this we will first introduce two types of contexts which can be identified in the right-hand-side. Then we will see how to use them.

- *Immediate context*: upon entering the rule, nodes in an immediate context may be evaluated to head normal form immediately.
- *Postponed context*: upon entering the rule, nodes in a postponed context may not be evaluated, and must be built as graphs, which can be passed as arguments to other rules, or given as a result.

We will call nodes immediate or postponed according to their context. The top node of a right-hand-side is an immediate node. All subnodes of a postponed node are postponed. The symbol of an immediate node determines the context of its argument nodes. For a node with a rule symbol all strict arguments are immediate, all other arguments are postponed. For nodes with a constructor symbol all arguments are postponed. Strictness for user-rules is given by annotations, for delta-rules it is known by the compiler.

Consider the following rules, in which rule 'S' has one strict argument and rule 'NS' has one non-strict argument (the boxes are postponed contexts):

```
F1 x -> Cons (S ... ) Nil ;

F2 x -> +I (*I x 10 ) 20 ;

F3 x -> +I (S x ) (NS ... ) ;

F4 x -> IF (=I (S x ) (NS ... )) (S x ) (NS ... ) ;
```

postponed contexts

In principle we have to build the right-hand-side graphs, as they are. However, if we discover an immediate node, while building the right-hand-side, we will not allocate it in the heap, but try to reduce it first and use the result. For user-rules this means calling the reduction code, for delta-rules the appropriate instructions are executed. If the top node of a right-hand-side contains a function symbol, the user rule will always be called (the top node is always immediate!). In the code we change this to a jump to the rule. This way we automatically remove tail recursion. For example:

```
F x -> F (…argument…);
```

tail recursion

`F` will actually be a loop in the generated code.

Things are less straightforward when we introduce sharing and cycles. We will not discuss the solutions here. We were able to devise a compilation scheme to cover all possible combinations of sharing and cycles in right hand sides, with the above principles.

6.4.6 SMALL STRICTNESS ANALYSIS

Although we consider strictness annotations to be generated by the compiler generating Clean, we incorporated a very simple strictness analyzer in our compiler. This analysis is based on certain aspects of the functional strategy.

Consider a rule with a pattern at the left-hand-side. Upon entering the rule we will always evaluate the actual argument for the first pattern. At compile time it is undecidable whether we have to match any of the other patterns in this rule, because the first match may fail. Therefore it is only the first argument with a pattern in a rule that can be marked as strict.

For example (the strict arguments are surrounded by boxes):

```
F1 x y:(Cons a b) z -> ………;

F2 Nil              -> ………|
F2 x                -> ………;

F3  x 10            -> ………|
F3 10  y            -> ………;

F4 Nil Nil Nil      -> ………;
```

strict arguments

6.4.7 EFFICIENCY OF THE GENERATED CODE

The compiler we constructed is slow, due to the fact that flexibility of the compiler was more important than compilation speed. The speed of the generated code, on the other hand, was of primary importance. The optimisations we devised are very suitable for VAX-like machines (PDP, MC68000). For other machines they may not always be the best. To get an impression of the speed of the code generated by the current implementation one can look in appendix B where some benchmark results are shown. Although these results show that lml is an order of magnitude faster then Clean, we may conclude that we are on the right track. Specially when we bear in mind that not yet all of the possible optimisations are included in the current Clean implementation. For example, leaf nodes are always built in the heap while the values could often be maintained on a stack.

6.5 CONCLUSIONS AND FUTURE RESEARCH

Clean is an experimental language with many facets. First of all it is a language for specifying computations in terms of graph rewriting. As such it is a convenient and elegant language.

Clean also has a very interesting underlying model of computation: a Graph Rewriting System which can be seen as an extension of a Term Rewriting System (Klop (1985)). This has the advantage that a lot of theoretical properties from the TRS world are inherited and provide a sound foundation for a GRS theory. For instance, in Barendregt *et al.* (1987a) it is proven that all hyper-normalizing strategies in the TRS world, a class to which all well-known normalizing strategies belong, are also normalizing in the GRS world.

Clean can be used as intermediate language between functional languages and (parallel) machine architectures. In (Koopman & Nöcker (1988)) it is shown that functional languages like SASL (Turner (1979a)), Miranda (Turner (1985)), OBJ2 (Futatsugi (1985)) and Tale (Barendregt & van Leeuwen (1986)) can easily be compiled to Clean code. Compilers (one written in Modula2, one written in Miranda) are being implemented targeted to Clean. With the current Clean implementation they run 30 to 50 times faster than the current Miranda system. The Clean implementation described in this paper runs reasonably fast considering the fact that we did not want to spend much time on trivial, but time-consuming, ad hoc optimisations (see appendix B).

Our plans are to improve Clean in the near future. We will do this in the more general Lean framework (Barendregt *et al.* (1987b)) in which Clean will be one of several possible subsets with certain desired properties (in this case geared to functional languages and suited for parallel architectures). Our intentions are to include separate compilation, modularization, general type system, unification, general IO etc. All this must be accomplished without loosing the basic elegance, the practical usability and the theoretical framework of the model. This will take some time.

Because strategies have a critical influence over efficiency future versions of Clean aim to give the programmer explicit control over rewrite order, for instance via high level specification of (parallel) reduction strategies and a formalism for mixing several strategy schemes during evaluation (van Eekelen & Plasmeijer (1986)).

We will improve the efficiency of the compiler and the code generated by the compiler. Implementations of Clean are planned for Motorola based architectures and parallel architectures like the Experimental Parallel Reduction Machine (Hartel & Vree (1986)) and the Distributed Object Oriented Machine (Odijk (1985)) being developed in the Philips Laboratories, the Netherlands. Requests for the current implementation can be sent to one of the authors or E-mailed to: *…!mcvax!hobbit!cleanrequest.*

6.6 ACKNOWLEDGEMENTS

We are grateful to Henk Barendregt and Pieter Koopman of the University of Nijmegen for several suggestions and inspiring discussions. We also thank Ronan Sleep, John Glauert and Richard Kennaway of the University of East-Anglia very much for the fruitful collaboration on the Lean work, which heavily influenced Clean.

6.A APPENDIX A: CLEAN SYNTAX

Clean syntax:

```
CleanProgram     = { RuleGroup }
RuleGroup        = [ `STRATEGY' StrategyName `;'] Rule { `|' Rule } `;'
Rule             = Graph `->' Graph
                 | Graph `->' Redirection
Graph            = [ Annotation ] [ Nodeid `:' ] Node { `,' NodeDefinition}
Redirection      = [ Annotation ] Nodeid
NodeDefinition   = [ Annotation ] Nodeid `:' Node
Node             = Symbol { [Annotation] Term }
Annotation       = `{' AnnotationName { `,' AnnotationName } `}'
                 | ShorthandAnnotation
Term             = Nodeid
                 | [ Nodeid `:' ] Symbol
                 | [ Nodeid `:' ] `(' Node `)'
```

Clean name conventions:

```
Symbol               = FunctionSymbol
                     | ConstructorSymbol
                     | DeltaRuleSymbol
                     | TypeConstructor
                     | TypeDenotation
Nodeid               = (* Character sequence starting with a lower-case character  *)
FunctionSymbol       = (* Character sequence starting with a upper-case character  *)
ConstructorSymbol    = (* Character sequence starting with a upper-case character  *)
DeltaRuleSymbol      = (* A predefined delta rule name                             *)
AnnotationName       = (* Implementation dependent                                 *)
ShorthandAnnotation  = (* Implementation dependent                                 *)
StrategyName         = `Functional'
TypeConstructor      = `INT' | `REAL' | `CHAR' | `STRING' | `BOOL'
TypeDenotation       = 5,4.6e-3,'a',"a string\007",TRUE (* Examples *)
```

Some context sensitive restraints:

- Graphs are connected.
- Sharing of labels is not allowed in left hand sides of rules.
- Symbols have a fixed arity.
- Every function is defined once.

- Every label is defined once in a rule.
- Delta rules can not be re-defined.

6.B APPENDIX B: PERFORMANCE MEASUREMENTS

The results of two benchmarks are reproduced here to give an idea about the speed of the compiled code. Benchmark 1 involves the reversion of a list, benchmark 2 is the all time favorite nfib number. The reversion benchmark reverses a list of n elements n times, this means doing n^2 reversion steps. In our tests n ranged from 1 to 1000. The nfib benchmark gives the number of function calls it did as output. We will only give the Clean programs here, it is straightforward to translate them to other languages* .

```
Reverse n            -> Walk (Rev_n n (FromTo 1 n));

Walk (Cons x Nil)    -> x      |
Walk (Cons x r)      -> Walk r;

Rev_n 1 list         -> Rev list Nil                |
Rev_n n list         -> Rev_n (--I n) (Rev list Nil);

Rev (Cons x r) list -> Rev r (Cons x list) |
Rev Nil list         -> list                ;
```

benchmark 1, Clean version.

```
Nfib 0 -> 1                                        |
Nfib 1 -> 1                                        |
Nfib n -> ++I (+I (Nfib (--I n)) (Nfib (-I n 2)));
```

benchmark 2, Clean version.

The following programming systems were tested:

* `++I` and `--I` are delta rules for integer increment and decrement, `-I` is for integer subtraction.

Clean	Clean Compiler, version 4.0, University of Nijmegen, Netherlands. Authors: Tom Brus, Maarten van Leer.
lisp	Franz Lisp interpreter, Opus 38.79, Unix 4.2 BSD distribution. Author: Keith Skowler.
liszt	lisp compiler, VAX version 8.36 [.79], Unix 4.2 BSD distribution. Author: John Foderaro.
lml	lml compiler, preliminary version, Chalmers, Sweden. Author: Lennart Augustsson, Thomas Johnsson.
miranda	miranda interpreter, version 0.292, Research Software Ltd., England. Author: David Turner.
saslcom	sasl compiler, version 1.1, University of Nijmegen, Netherlands. Author: Riet Oolman.
saslint	sasl interpreter, version 1.1, University of Nijmegen, Netherlands. Author: Riet Oolman.

All tests were done on a VAX11/750 under UNIX BSD 4.2, partly during working hours. All times mentioned are user times returned by the time(1) command. We measured the number of reverse steps per second (for reverse), and the number of function calls per second (for nfib):

We see that these numbers stabilize to what we call the reverse number and the nfib number of the implementation. Below, these numbers are tabled separately.

language	rev number	nfib number
saslint	8	7
mira	123	120
lisp	151	467
saslcom	677	728
liszt	1669	1258
clean	3521	2322
lml	23436	19635