Dactl: An Experimental Graph Rewriting Language*

J.R.W.Glauert , J.R.Kennaway and M.R.Sleep

Declarative Systems Project, UEA, Norwich NR4 7TJ, U.K.

1 Introduction.

Term (or tree) rewriting systems have proved useful both as specifications and — though less commonly — as practical systems for symbolic computation (see [HO82] for a practical system with a sound theoretical underpining). Klop [Klo90] and Dershowitz and Jouannaud [Der89] provide comprehensive treatments of term rewriting theory, which is now reasonably well understood.

The idea of studying transformation systems based on graphs (as opposed to trees) dates back at least to [Ros72], and a significant body of theory has been developed, most notably by the Berlin school of Ehrig and others: [Ehr89] gives an authoritative overview.

Practical uses of 'graph rewriting' date back at least to Wadsworth [Wad71], which develops a graph based representation of lambda terms and an associated implementation method for normal order evaluation of lambda calculus expressions. The relation between tree and graph rewriting has been studied in some detail [Sta80a, Sta80b, Bar87, Hof88, Far90]. The main result is that sharing implementations produce the correct semantics at least for *orthogonal* term rewrite systems[1].

New generation Logic languages of the committed choice variety (for example Concurrent Prolog [Sha86] and Parlog [Gre87]) may be viewed as specialised graph rewriting languages, as may actor models such as DyNe [Ken85]. More recently Lafont [Laf89] has proposed an interaction net model of computation which again may be viewed as specialised graph rewriting, whose constraints are inspired by Girard's work on Linear Logic.

In 1983 the authors undertook an ambitious project aimed at designing a common model of computation which would be general enough to support a range of more restricted computational models such as those required for functional, logic and actor-like languages. The primary aim of the project was to produce a common target language (CTL) for a range of symbolic processing languages, particularly functional languages and committed choice logic languages. The project chose *graph rewriting* as the basis for its work.

The main success of the project was the design and implementation of a general model of computation based on graph rewriting. The model is called DACTL (for Declarative Alvey Compiler Target Language). The main failure of the project was that it proved difficult within the timescale to develop the compiler technology necessary for Dactl to act as an *efficient* CTL: we seriously underestimated the work needed here. Nevertheless, it was possible to demonstrate working compilers for a surprisingly wide range of languages including HOPE, LISP, PARLOG, GHC, ML and CLEAN within the timescale [Ham88,Gla88a,Gla88b], [Ken90a].

The CTL motivations of the Dactl project are now mainly historical. What remains is one of the

---

* This work was partially supported by ESPRIT project no. 2025 (European Declarative System) and basic research action no. 3074 (Semagraph).

[1] An orthogonal Term Rewrite System is both left linear, and non-overlapping.

few genuine graph rewriting language implementations in existence. There is a stable, reasonably engineered implementation of Dactl for the Sun with modular compilation facilities and a comprehensive Unix interface, and a more recent implementation for Macintosh computers which is in regular use. Our experience of the language design process, together with our experience in using Dactl in its present form suggest that others may find it useful as an experimental tool for exploring practical graph rewriting systems.

The paper is organised as follows. The remainder of this introductory section outlines the main features of Dactl, and briefly describes the history of the Dactl project. The body of the paper consists of a more detailed description of Dactl, and a variety of illustrations of its use. Finally, the relationship between the operational semantics of a Dactl rewrite and categorical semantics of graph rewriting is briefly discussed.

1.1 Main features of the Dactl language.

a. Dactl graphs are *term graphs* in the sense of [Bar87][1]. That is, every node has a symbol (or label) together with zero or more directed out-arcs to other nodes. Thus Dactl nodes together with their symbols and out-arcs correspond to the labelled *hyperedges* used to model term graph rewriting in the Jungle evaluation model developed more recently by Hoffmann and Plump[Hof88].

b. A Dactl rewrite is *atomic*. This is expressed by requiring that every valid outcome of a Dactl computation must correspond to an outcome which could be reached by sequential execution. The great benefit of atomicity is that invariance of properties across individual rules also holds for all valid executions. The cost is that an implementation must ensure that co-existing conflicting rewrites are not executed concurrently. This may be done for example by locking critical nodes. For certain classes of rule systems, it is possible to show that no locking is needed to ensure the correctness of concurrent execution of rewrites[Ken88].

c. A Dactl rewrite may contain a multiple reassignment of out-arcs (called *redirections* in Dactl terminology). It is this feature which gives Dactl much of its expressive power, allowing non-declarative behaviours to be expressed.

d. Dactl graphs include *control markings* on the nodes and the arcs. These allow a wide range of evaluation strategies and synchronization conditions to be expressed. The control markings are an integral part of Dactl: a graph which contains no control markings is not rewritable according to Dactl semantics, even if the graph contains redexes in the usual TRS sense. Techniques for generating appropriate markings automatically are reported by Kennaway [Ken90a], and Hammond and Papadopoulos [Ham88].

e. Dactl supports separate compilation, and a classification scheme for symbol usage which allows the writer of a Dactl module to constrain external use of exported symbols by appropriate symbol class declarations.

f. The implementation supports a comprehensive interface to Unix.

g. The implementation gathers statistics and execution traces corresponding to both sequential and parallel execution.

Whilst it is clear that the design could be improved, we believe that this is best delayed until there is significantly more experience with the present design. The current definitive reference document for Dactl is Final Specification of Dactl [Gla88c], obtainable from the authors.

1.2 Project History.

In 1983 a number of ad-hoc meetings were held in the U.K. in an attempt to identify a common basis for the development of parallel machines suited to the needs of the ‘new generation’

---

[1]In fact Dactl graphs are more general than those arising naturally from terms: Dactl graphs may be cyclic, whereas 'term graphs' are DAGs.

languages, particularly those based on logic formalisms (the *logic* languages) and those based on the lambda calculus (the *functional* languages).

In April 1984 the U.K. Alvey directorate sponsored a meeting at the Royal Society of key workers from academia and industry to consider a proposal to develop a common model of parallel computation. The meeting identified strong polarization between those who believed that efficiency was paramount, and those who believed that the benefits of working towards a common model outweighed potential performance drawbacks. It was recognised that many specialist parallel architectures would evolve which required specialist interfaces The outcome of the meeting was a decision to proceed with a project whose aims were limited to identifying a common model for 'declarative' languages.

By May 1985, work had reached the stage where a preliminary proposal for such an interface could be given limited circulation. This was followed in September 1985 by the first release of a reference interpreter for the preliminary version of the interface, which was given the title Dactl (for Declarative Alvey Compiler Target Language).

During this period a consortium involving ICL, Imperial College, Manchester University and East Anglia was formed to develop this early work in the context of the Alvey Flagship[Wat87] project which focussed on declarative languages and parallelism. Alvey funded work on Dactl, based at UEA, began in May 1986. The technical aims of the work were primarily concerned with developing a precisely defined graph rewriting model of computation and exploring its properties, working closely with the Flagship team. The development of the model was to be expressed as a series of reports defining the model, together with a number of releases of reference interpreters.

Work on the more formal aspects was aided by collaboration with the Dutch Parallel Reduction Machine Project led by Prof.H.P.Barendregt. This collaboration led to a number of joint publications[Bar87,Bar89], including a paper specifying a common abstract model of graph rewriting called LEAN (which, apart from syntax, is essentially Dactl without the control and synchronization markings).

This formal work with the Dutch, together with modified requirements input from the Flagship team, led to a fundamental redesign of the computational model. A specification of the core model resulting from this work was released in March 1987 (the Core Dactl report) and accepted by Flagship shortly afterwards. A preliminary definition of a revised design of Dactl was completed in June 1987. This was augmented by a release of the design in December 1987 which included a very detailed UNIX interface.

The final design of the language was released early in January 1989, and a consistent supporting version of the reference interpreter followed shortly afterwards.

2 The Dactl Graph Rewriting Model of Computation.

We start by considering the canonical representation of graphs used by Dactl, and then describe the form of rewriting rules in the language. Later we discuss the details of rewriting and control of the rewriting process.

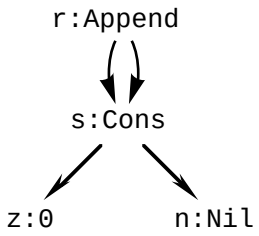
2.1 Dactl Graphs

A Dactl program manipulates *directed graphs.* Each *node* is labelled with a *symbol* which may be interpreted as a function, predicate, or constructor according to the requirements of the computation being implemented. From nodes will originate an *ordered* sequence of zero or more directed *arcs* leading to *successor nodes*. Graphs may be cyclic and need not be connected, but there is a distinguished node in the graph known as the *root*.. When considering the final form of a graph, only nodes reachable from the root are (by definition) of interest. Hence unreachable nodes which cannot affect the final form may be removed from the graph along with their successor arcs.

The following examples give the textual and pictorial representation of two graphs.

Example 1: A DAG

Pictorial form:



Shorthand textual form: `Append[s:Cons[0 Nil] s]`

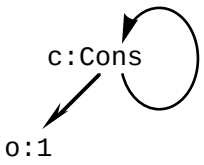
Equivalent longhand textual form:

```
r: Append[s s],
s: Cons[z n],
z: 0, n: Nil
```

The longhand form is a tabular representation of the graph. A node is made up of a symbol and a list of the identifiers of successor nodes. Each node is given an identifier beginning with a lower-case letter. Integers and identifiers beginning with an upper-case letter represent symbols. The root of the graph is taken to be the first node specified. In the shorthand form we may combine the definition of a node with one of its occurrences and may omit unnecessary node identifiers. Note that a graph equivalent to a ground term may be described without using node identifiers.

Example 2: A Cyclic Structure

Pictorial form:



Shorthand textual form: c: Cons[1 c]

Equivalent longhand textual form: c: Cons[o c], o: 1

2.2 Dactl Rewriting Rules

The reduction relation for a Dactl system is described by a set of rewriting rules which describe *graph transformations.*

The left-hand side of a rule consists of a *pattern* which is a generalisation of a Dactl graph. Any Dactl graph as described above is a Dactl pattern. In addition, a pattern may contain special *pattern symbols*, which match a class of symbols, and *pattern operators*. The simplest special pattern symbol is ANY, which identifies a variable node. The pattern operators of Dactl are +, - and & and represent union, difference and intersection respectively.

Before rewriting can take place, it is necessary to establish a *match* between a subgraph of the program graph called a *redex*, and the pattern of a rule. Formally, this means identifying a structure preserving mapping between the nodes on the pattern and the graph undergoing rewriting.

The right-hand side of a rule includes the *contractum graph*, and a number of *redirection*s. Rewriting involves building the contractum, a copy of the right hand side of the rule, and connecting it into the original graph according to the *redirections* specified as part of the rule. Very frequently only a single redirection of the root is intended, and the syntax of Dactl provides a special connective => between the left and right hand side of a Dactl rule for this purpose.

The following example rules model the appending of lists. In shorthand form, the rules are:

```
Append[Cons[h t] y]   =>   Cons[h Append[t y]]  |
Append[Nil y]   =>   y
```

Apart from the square brackets and the => and | symbols, these rules take the form of term rewrite rules. It is a principle of Dactl design that the meaning of all shorthand is given by translation to longhand canonical form. For the above rules, this involves a tabular listing of the nodes in the pattern and the contractum, and the explicit inclusion of redirections, which take the syntactic form of conventional assignment statements.

```
r: Append[c y], c: Cons[h t], h: ANY, t: ANY, y: ANY
      ->   s: Cons[h b], b: Append[t y], r:=s   |
a: Append[n y], n: Nil, y: ANY   ->   a:=y
```

2.3 Graph rewriting in Dactl.

The general form of a (longhand) Dactl rule is:

```
Pattern     ->    Contractum, Redirections, Activations
```

A single Dactl rewrite takes place in four phases, namely *match*, *build*, *activate*, and *redirect*..

The *match* phase identifies a graph homomorphism — that is, a structure-preserving mapping — between the pattern and the graph undergoing rewriting.

The *build* phase adds new nodes (as specified by the contractum) to the graph. Where the contractum contains occurrences of variable nodes these are replaced by their bindings

The *redirect* phase performs redirections which change the destination of some arcs of the original graph, allowing the 'gluing in' of new contractum nodes into the existing graph.

The *activate* phase adds active node control markings to the nodes specified. This allows a rule to propagate zero or more control loci.

These phases are described in more detail below.

2.3.1 Matching

A match is a graph homomorphism from the pattern of a rule to the program graph. Structure is preserved by this mapping, except at variable nodes. There is a match between the pattern of the rule:

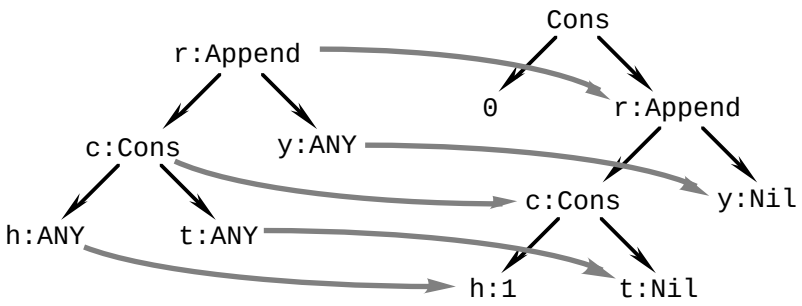
```
r: Append[c:Cons[h:ANY t:ANY] y:ANY]  =>  s:Cons[h b:Append[t y]]
```

and the following example graph:

```
Cons[0 r:Append[c:Cons[h:1 t:Nil] y:Nil]]
```

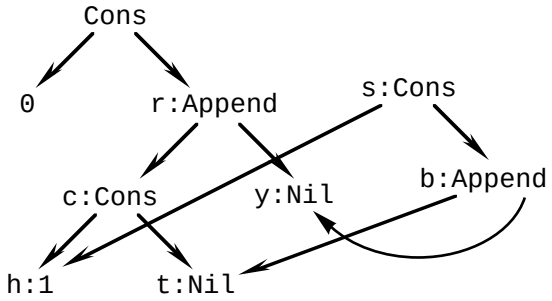
The subgraph of the program graph onto which the nodes of the pattern are mapped is known as the *redex*. The root of the pattern has a special significance. We say that a pattern *matches at a node* if the node in question is the root of a redex for the pattern. The matching process identifies bindings for the variable nodes in the pattern: in the example given, the binding is {y–>y, h–>h, t–>t} and takes this simple form because of the careful choice of node identifiers in the subject graph.

Note that the atomic rewriting principle of Dactl allows us to ignore the operation of other parallel rewriting activities in the graph in describing a rewrite.



2.3.2 Building

The second phase of rewriting builds a copy of the contractum of the rule matched. The contractum contains no pattern symbols, but may contain occurrences of identifiers from the pattern. During building, such occurrences become arcs to the corresponding nodes matched in the first phase. After building, the example graph has the form:




```
Cons[0 r:Append[c:Cons[h:1 t:Nil] y:Nil]],
s: Cons[h b:Append[t y]]
```

where the new graph is on the second line. The build phase is missing for ‘selector’ rules, such as the second rule of `Append`.

2.3.3 Activation

There are two means of propagating control loci in Dactl. One means is to specify some of the new nodes in the contractum as active: this mechanism is sufficient for many applications.

The second means is to alter the control state of a pre-existing node in the graph undergoing rewriting, and which has been matched to a node variable in the pattern. This is the function of the activate phase of Dactl rewriting. Operationally it can be thought of as following a reference acquired during the match phase, and changing the control marking of the relevant node to indicate that a rewrite is to be attempted by the execution mechanism.

There is no implicit priority mechanism in Dactl: the execution mechanism may select any active node available, and it is the responsibility of the Dactl programmer to program in any notion of ‘fairness’.

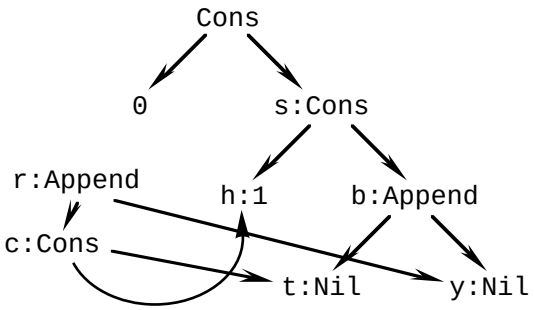
2.3.4 Redirection

The build phase allows the new portion of the graph to contain references to parts of the subject graph. The purpose of the redirection phase is to allow references in the old graph to be changed consistently to refer to parts of the new structure. Very general transformations are possible, as any or all of the nodes identified by pattern variables may be redirected within a single atomic rewrite[1].

Following a rewrite according to the first rule of the Append example, we expect to find references to the new `Cons` node, `s`, in place of references to the `Append` node. In other words, we wish all arcs referencing the root node, `r`, to be *redirected* to reference `s`. Hence, all occurrences of `r` as a successor of another node are replaced by `s`. The resulting graph, including all the ‘garbage’ not reachable from the root, is:

[1]Whilst very general multiple redirections are allowed in Dactl, there are restrictions designed to ensure that a Dactl rewrite is well defined. These restrictions very roughly correspond to the gluing conditions in the classical double pushout model.

```
Cons[0 s:Cons[h:1 b:Append[t:Nil
y:Nil]]],
r: Append[c:Cons[h t] y]
```



In this case the original Append node may be garbage collected. A common implementation technique is to *overwrite* the Append node with the contents of s, thus avoiding in practice the overhead of supporting genuine physical redirection of pointers.

Performing the redirections is the final phase of rewriting. Nodes from which arcs are redirected are always nodes from the original graph. The target of the redirection may be in the original graph, or in the contractum as in our example. The graph is left in a state where a redex for the second rule exists. As before, we give the rule:

```
a: Append[n:Nil y:ANY] => y
```

and the graph, showing the match:

```
Cons[0 Cons[1 a:Append[n:Nil y:Nil]]]
```

No graph is built in the redirection phase of rewriting. The right hand side of the rule indicates that the graph should be rewritten so that in place of the Append node at the root of the redex, we now see the node referenced by the second argument, y. This parallels term rewriting theory in which, following a corresponding rewrite, the original parent of a would find the subterm rooted at y as its direct descendant in place of a. The effect of replacing all references to a by y is the following graph:

```
Cons[0 Cons[1 y:Nil]], a:Append[n:Nil y]
```

No references to the Append node will now remain. It is common for implementations to re-use the node by changing it to an *indirection* to y.

2.3.5 Pattern Operators

The Dactl rules illustrated so far are left-linear: the patterns can be expressed in shorthand form with no repeated node identifiers. However, Dactl *does* allow repeated pattern identifiers in which case each occurrence must match the same program graph node. This interpretation arises naturally from the definition of matching in terms of a graph homomorphism.

In addition to the pattern symbol ANY, Dactl also provides three pattern operators. These are expressed in infix notation. If Π and Σ are patterns, then the following are also patterns:

```
(Π + Σ)     (Π - Σ)     (Π & Σ)
```

A pattern can be regarded as defining a set, being the set of all Dactl graphs which will match the pattern with the root of the graph being the root of the redex. The first two pattern operators act like union and difference operators on such sets. Hence, the first form of pattern matches at a node if either Π or Σ matches at the node and the second requires that Π matches, but Σ does not. The & operator corresponds to set intersection.

These pattern operators will not be considered in detail in this paper, but they prove useful in restricting cases where more than one rule may apply. For illustration, we give rules for the ubiquitous factorial function in which we wish to exclude graphs matching the first rule from matching the second:

```
Fac[ 0 ] => 1     |
Fac[ n:(INT-0) ] => IMul[ n Fac[ ISub[n 1] ] ]
```

Nearly all practical term rewriting languages are designed as priority rewrite systems [BEG87], in which the textual ordering of the rules expresses rule priority. In Dactl it is possible to disambiguate such rule systems by careful use of pattern expressions, and Dactl is unusual in this respect.

However, priority rewrite semantics is both common and convenient, and so syntactic sugar is provided in Dactl to support it. Rules separated by a semi-colon are matched in order, whereas

rules separated by | may be dealt with in any order.

```
Fac[ 0 ] => 1     ;
Fac[ n:INT ] => IMul[ n Fac[ ISub[n 1] ] ]
```

There is an implicit use of the pattern difference operator to exclude graphs matching the pattern of the first rule from matching the second rule. The obvious implementation, which has significant performance benefits, is to consider the rules in the given textual order.

The graph rewriting framework, based on pattern-matching rules, gives great expressive power to Dactl. *Ambiguity* required by languages allowing non-deterministic results is expressed by systems in which redexes overlap. When disjoint redexes exist in a graph, *concurrency* can be exploited by an implementation. Imperative and Object-oriented code which manipulates program *state* can be implemented using rules which redirect references to nodes not at the redex root. This technique can also be used to implement logic variables. The graphical basis enables *sharing* of subterms to be expressed and exploited.

2.4 Control of evaluation in Dactl.

Dactl is concerned with control of evaluation as well as the properties of an abstract rewriting system. During the design of Dactl we observed that differences in evaluation strategy markedly affect both the ‘look and feel’ of a given language, and also the semantics. Even the more modern ‘lazy’ functional languages include some operational rules about the way in which pattern matching is handled in their semantics. Laville [Lav87], Kennaway [Ken90c] and others have examined this problem, but in terms of language design the expression of strategy and control remains a subtle problem. This is partly a human factors problem of course, and hence not amenable to theory in its present state.

Faced with these problems we decided to include fine grain control and synchronization markings as an integral part of Dactl. It was recognised at the time that contemporary technology for parallel computing was easiest to exploit using coarse grain parallelism: our work on the ZAPP architecture[McB87] did just that in the context of transputers. But technology advances, and fine grain parallelism may look much more realistic quite soon.

The intuitive basis of Dactl’s control markings is that each agent executing a Dactl rule is a *locus of control* of some process. In the von Neumann model, there is exactly one process and exactly one locus of control. Each instruction appoints a unique successor. In a Dactl rule, zero or more successor ‘control loci’ may be appointed either by creating new nodes active during the build phase, or by activating nodes in the original graph during the activate phase. The single syntactic token * serves for both purposes in Dactl.

An active node can be thought of as a process, and processes need to communicate and synchronise. This leads to Dactl’s notification markings on arcs, which specify reverse communication paths in the graph, and suspension markings on nodes to enable the expression of processes suspended awaiting a certain number of events.

A fundamental design question for Dactl was ‘when should rewriting agents communicate?’. The decision taken was to adopt a single, very simple rule:

Dactl rewriting agents communicate when they attempt a match at an active node, and the match fails.

Intuitively, failure to match indicates that some sort of temporally local normal form has been reached: it’s not in general a normal form, because future Dactl rewrites may make it a redex. But failure to match at a node is a key event, worth signalling to all who have marked references to the node and this is the principle adopted in Dactl.

2.4.1 Dactl Markings

Dactl encodes the strategy in a pattern of *markings* on the nodes and arcs of the program graph. There are two forms of node marking and a single form of arc marking:

The most important *node* marking is the *activation* denoted by *****. *Only activated nodes are considered as the starting points of rule matching*. Once a Dactl program graph contains no activations, execution is complete and the graph viewed from the root is the result of computation. It may, or may not, be in normal form with respect to the rewrite rules stripped of markings.

The marking **#** indicates a suspended node, and a node may have one or more such markings enabling it to await a specified number of notification events before becoming active again. The node concerned will not be considered for matching immediately, but will be reconsidered when the corresponding number of notifications is received.

The arc marking **^** is used in conjunction with # for such synchronization purposes. It indicates a notification path between the target of the arc and its source. When evaluation of the target is complete in the sense that rule matching fails (for example because it has been redirected to a node with a constructor symbol) the arc marking is removed along with a # marking on the source node, if present. When the last # is removed, it is replaced by *****, thus making the node active. This supports a model of evaluation close to dataflow since operators wait until their operands are available before being rewritten.

The operator ***** may also be used on the right hand side before the identifier of a node matched by the pattern. This is taken as an instruction to activate the corresponding node if it is currently neither active nor suspended.

All the markings, including both uses of ***** are illustrated by the following version of the Append rules:

```
Append[Cons[h t] y] => #Cons[h ^*Append[t y]]   |
Append[Nil y] => *y
```

2.4.2 Dactl Rewriting with Markings

If matching succeeds at an active node, the ***** marking is removed, the contractum is built, required nodes are activated, and redirections are performed. Most rules redirect references to the original root node which may then be garbage collected.

The criterion for notification is *failure to match* an active node to any rule. The activation marker is removed from the node and any notifications required by direct ancestors are performed. Matching failure most commonly occurs with constructor nodes for which there are no rules. To return a result therefore, a rule will usually redirect its root to an activated constructor node. Considering the marked rules given above with a new graph of the form:

```
a:*Append[k:Cons[o:1 n:Nil] k]
```

We see that there is a redex for the first rule with the node k in the graph matched by more than one part of the pattern. It is perfectly consistent for a tree-structured pattern to match a graph with sharing. After rewriting, the structure is:

```
m:#Cons[o:1 ^b:*Append[n:Nil k:Cons[o n]]], a:Append[k k]
```

The original node a is now garbage and can be removed. There is now a redex for the second rule and the graph is rewritten to:

```
m:#Cons[o:1 ^k:*Cons[o n:Nil]], b:Append[n k]
```

The node b is now garbage. Evaluation is now complete, but notification of the ancestors of an Append node is delayed until the whole operation is complete. This is achieved by suspending the Cons node m waiting for evaluation of the rest of the list. Cons is a constructor so matching fails and the parent node, m, is notified and hence activated:

```
m:*Cons[o:1 k:Cons[o n:Nil]]
```

Again, the Cons node, m, will match nothing, so the final graph will be:

```
m:Cons[o:1 k:Cons[o n:Nil]]
```

The examples used display no concurrency during evaluation. Concurrency arises when the right-hand side of a rule contains several active nodes so that the rule nominates many successors to receive control. Also, several nodes may be suspended awaiting notification from

the same node, and all will be activated once it is evaluated.

3 Some Dactl Examples.

3.1 Sorting.

```
MODULE SortModule;
IMPORTS Arithmetic; Logic; Lists;
SYMBOL REWRITABLE PUBLIC CREATABLE Sort;
SYMBOL REWRITABLE Insert;Compare;
RULE
      Sort[x:Nil]  =>  *x;
      Sort[Cons[h t]]  =>  #Insert[h ^ *Sort[t]];

      Insert[n x:Cons[h t]]  =>  #Compare[ ^*IGt[n h] n x];
      Insert[n Nil]  =>  #Cons[ ^ *n Nil];
      Insert[n x:(ANY-Nil-Cons[ANY ANY])]  => #Insert[n ^ *x];

      Compare[True n x:Cons[h t]]  =>  #Cons[h ^*Insert[n t]];
      Compare[False n x]  =>  *Cons[n x];
ENDMODULE SortModule;
```

The following module illustrates the use of the sort module.

```
MODULE SortTest;
IMPORTS SortModule;
RULE
      INITIAL => *Sort[Cons[5 Cons[2 Cons[9 Cons[3 Cons[1 Nil]]]]]];
ENDMODULE SortTest;
```

3.2 A simple Head Normal Form reducer for Combinatory Logic

We use explicit binary application, and define patterns for HNF (Head Normal Form) and also for redexes. This allows us to write just 3 rules, two for the redex cases and one for the (ANY–Redex–Hnf) case.

```
MODULE SK1;
SYMBOL REWRITABLE PUBLIC CREATABLE Ap;
SYMBOL CREATABLE PUBLIC CREATABLE S; K;
PATTERN PUBLIC Hnf = (S+K+Ap[(S+K) ANY]+Ap[Ap[S ANY] ANY]);
PATTERN Redex = ( Ap[Ap[K ANY] ANY] + Ap[Ap[Ap[S ANY] ANY] ANY] );
RULE
      Ap[Ap[K x] y]  =>  *x;
      Ap[Ap[Ap[S f] g] x]  =>  *Ap[Ap[f x] Ap[g x]];
      (Ap[x y]&(ANY-Redex-Hnf))  =>  #Ap[ ^x y], *x;
ENDMODULE SK1;
```

If we activate a combinatory term in the presence of these rules, it will notify if and when it reaches a head normal form. For example,

```
*Ap[ Ap[ Ap[ S S ] K ] Ap[ Ap[ K S ] K ] ]
```

(representing the combinatory term which would conventionally be denoted by SSK(KSK)) will when evaluated by these rules notify when it reaches the form

```
Ap[ Ap[ S x ] Ap[ K x ] ], x:Ap[ Ap[ K S ] K ]
```

(representing the term S x (K x) where x=KSK). To obtain a normalising reducer we add the following rules:

```
Ap[K x]  –>  *x;
Ap[S x]  –>  *x;
Ap[Ap [S f] g]  –>  *f, *g;
```

The left-hand sides of these rules together match all terms in head normal form, and only such terms. They do not do any redirection, but merely activate their components. The effect is to obtain the normal form of any term which has one; however, the parents of such a computation

will not be notified when it has completed. A notifying solution can be obtained at the expense of slightly complicating the rule set.

3.3 Graph Copying.

It is something of a surprise to newcomers to graph rewriting that graph copying is not automatically a primitive operation. The primary reason is that graph rewriting models such as Dactl seek primitives which are bounded at least by the size of rule. Provided such primitives can be used to express copying as a graph reasonably efficiently with respect to the primitives, the lack of a graph copy primitive is not seen as a cause for concern. We give two solutions to the graph copying problem.

3.3.1 Non-notifying solution.

Here is a simple Dactl program which copies a graph constructed from the symbols Pair and Leaf. It creates two copies of a graph, but does not restore either copy to the original context, nor does is notify when the copying operation is complete.

```
MODULE CopyGraph;
SYMBOL GENERAL Leaf;Pair;L;R;Copy;D;
RULE
      INITIAL  =>  *Copy[r:Pair[s:Pair[t:Pair[u:Pair[r t] s] r] u]];
      Copy[r:Leaf]  ->  r':D[Leaf Leaf], r:=r';
      Copy[r:Pair[a b]]  ->  r':D[*Pair[a b L] *Pair[a b R]], r:=r',
                             *Copy[a], *Copy[b];
      Pair[D[ll lr] D[rl rr] L]  =>  Pair[ll rl];
      Pair[D[ll lr] D[rl rr] R]  =>  Pair[lr rr];
ENDMODULE CopyGraph;
```

The essential feature of the program is to perform a recursive scan of the graph from the root, replacing all references to the nodes encountered by references to copies 'guarded' by the symbol D. Besides inheriting all references to the original node, the guard acts as a stopper for the copying processes in the case of circular graphs. The two copies of a Pair node made under a D guard are distinguished by the additional symbols L or R, thus exploiting Dactl's freedom to use the symbol Pair with differing arities (2 and 3 in this case). The rules for the ternary use of the Pair symbol extract two distinct copies of the graph.

3.3.2 A full solution.

The solution above illustrates the basic principle of graph copying in Dactl, but fails to preserve a version of the original graph in context. The following solution does this, and in addition notifies when the operation is complete.

```
MODULE GraphRestore;
SYMBOL OVERWRITABLE  Nay; Yea;
SYMBOL REWRITABLE  CopyRestore; RestoreCopy; Copy; Restore; Sync;
SYMBOL CREATABLE  MPair; MLeaf; Hold;
RULE
      INITIAL  =>  Hold[ *CopyRestore[r] r s l],
                   r: MPair[s s:MPair[r l:MLeaf[Nay] Nay] Nay];
      CopyRestore[ g ]  =>  #RestoreCopy[ ^*Copy[g] g];
      RestoreCopy[n o]  =>  #Hold[n ^*Restore[o]];
      Copy[ MPair[l r n:Nay] ]  =>
            ##Sync[ ^ml:*Copy[l] ^mr:*Copy[r] y],
            n := Yea[ y:MPair[ml mr Nay] ];
      Copy[ MPair[l r Yea[p]] ]  =>  *p;
      Copy[ MLeaf[n:Nay] ] =>  y:*MLeaf[Nay], n := Yea[y];
      Copy[ MLeaf[ Yea[y] ] ]  =>  *y;
      Restore[p:MPair[l r y:Yea[n]]]  =>
            ##Sync[^*Restore[l] ^*Restore[r] p], y := Nay;
      Restore[ p:MPair[l r Nay] ]  =>  *p;
      Restore[ l:MLeaf[ y:Yea[n] ] ]  =>  *l, y := Nay;
```

```
        Restore[ l:MLeaf[ Nay ] ]  =>  *l;
        Sync[a b c] => *c;
ENDMODULE GraphRestore;
```

Note that this solution assumes that each node supports a housekeeping argument (set to Yea or Nay). This is not an unrealistic assumption in practical implementations.

4 Translating other languages to Dactl.

The translation of strongly sequential term rewrite systems to Dactl (complete with correct control markings) is described in [Ken90a]. Translation schemes for both functional and logic languages to Dactl are described in [Ham88]. Here we give some brief examples intended to illustrate the techniques used.

4.1 Strict Evaluation

We gave some rules for evaluating the Append The form of evaluation illustrated earlier for Append will fail to rewrite if the first argument has yet to be evaluated to a list. An extra rule of this form would ensure evaluation:

```
Append[x y]  =>  #Append[^*x y]
```

Although the first argument will be evaluated, rewriting may complete without evaluating the second argument to a list. To force the function to be strict, we should add rules to coerce the result to a list. The full set would be:

```
RULE
        Append[Cons[h t] y]  =>  #Cons[h ^*Append[t y]] |
        Append[Nil y]  =>  *ForceList[y]    ;
        Append[x y]  =>  #Append[^*x y]     ;
RULE
        ForceList[n:Nil]  =>  *n      |
        ForceList[Cons[h t]]  =>  #Cons[h ^*ForceList[t]]     ;
        ForceList[a]  =>  #ForceList[^*a]   ;
```

4.2 Lazy Evaluation

It is common to use a more lazy form of evaluation to *head-normal form.* Roughly, this means that evaluation proceeds until the outermost node of the expression is a constructor. The rules would be:

```
RULE
        Append[Cons[h t] y]  =>  *Cons[h Append[t y]]   |
        Append[Nil y]  =>  *y   ;
```

The recursive application of `Append` is not reduced by the first rule. The result of the second rule is activated, however, and should reduce to a constructor. A default rule could be employed in case the first argument was not a `Cons` or `Nil`. Techniques have been developed at UEA for the translation of the functional language Clean which ensure that arguments have always been evaluated sufficiently so that such default rules are not needed[Ken90a].

4.3 Early Completion

To allow for stream parallelism an early completion scheme can be used. In this, we notify when a head-normal form has been produced, but continue to evaluate to normal form at the same time:

```
RULE
        Append[Cons[h t] y] =>  *Cons[h *Append[t y]]   |
        Append[Nil y]  =>  *ForceList[y]    ;
        Append[x y]  =>  #Append[^*x y]     ;
RULE
        ForceList[n:Nil]  => *n |
        ForceList[Cons[h t]] =>  *Cons[h *ForceList[t]] ;
```

```
ForceList[a]  =>  #ForceList[^*a]   ;
```

4.4 Examples from moded Logic programming.

Translating flat concurrent logic languages to Dactl has been studied at UEA and by the Parlog group at Imperial College. We illustrate the flavour of these translations using an alternative technique for Append. The rule will attempt to instantiate the third argument with the result of appending the second argument to the first. The symbol Var[1] indicates an uninstantiated variable. This may be replaced by a data value by redirection of arcs to a *non-root* node. This is the basis of many techniques in Dactl for handling logic and object-oriented programming.

The rule set used would be as follows:

```
RULE
      root:Append[Nil y:(ANY-root) v:Var]  =>  *Succ, v:=*y |
      Append[Cons[h t] y v:Var]  =>  *Append[t y n:Var],
                                        v:=*Cons[h n]          |
      Append[x:Var y v:Var]  =>  #Append[^x y v]              ;
      Append[x y v]  =>  *Fail                                ;
```

In the first case the variable, v, is instantiated to the second argument. By firing it, we force a notification to be passed to any node suspended waiting for this instantiation. The second rule is similar, but we notify those awaiting v with a Cons node whose second argument is a variable which will be instantiated eventually by the evaluation of the new Append.

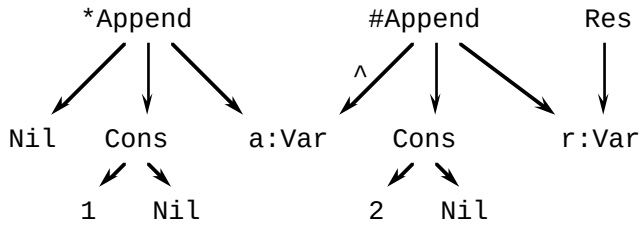
The third rule suspends on the variable, but does not activate it. Activation would be pointless since there are no rules for Var and notification would return immediately. However, notification will be provoked eventually by the rule which instantiates the variable. The final, failure case indicates that the first argument is some data value, but not a list, or that the third argument is not a variable. We will illustrate evaluation of the following graph:

```
Res[r], *Append[a Cons[2 Nil] r:Var], *Append[Nil Cons[1 Nil] a:Var]
```

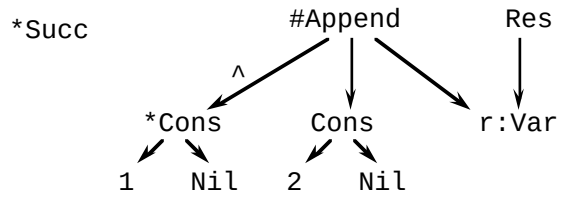
Taking the first active node:

```
=>    Res[r], #Append[^a Cons[2 Nil] r:Var],
      *Append[Nil c:Cons[1 Nil] a:Var]
```

The term suspends until the variable is instantiated:



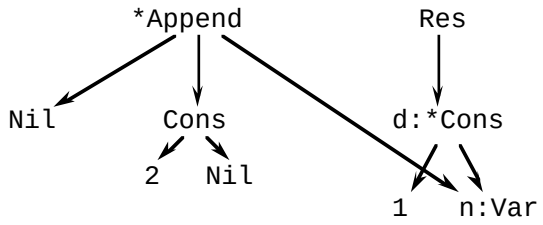

```
=>    Res[r], #Append[^c Cons[2 Nil] r:Var],
      *Succ, c:*Cons[1 Nil]
```



[1]Note that Var is not a special symbol in Dactl: we could have chosen other symbols for this example. It is the rule system, together with the graph rewriting semantics of Dactl, which make Var have some of the properties of a logic variable.

```
=>    Res[r],
      *Append[c:Cons[h:1 t:Nil] y:Cons[2 Nil] r:Var]

=>    Res[d], *Append[t:Nil y:Cons[2 Nil] n:Var],
      d:*Cons[h:1 n]
```




```
=>    Res[Cons[1 y]], *Succ, y:*Cons[2 Nil]
```

And Finally:

```
=>    Res[Cons[1 Cons[2 Nil]]]
```

5 Categorical Semantics.

The semantics of Dactl can be divided into two parts:

a. the semantics of an individual graph rewrite.

b. the semantics associated with Dactl's control markings.

In [Ken90b] one of us has presented a category-theoretic definition of graph rewriting in the category of jungles. It unifies the two previous categorical models of [Rao84,Ken87] and [Ehr79].. Our concern in that paper was to describe term graph rewriting, but in fact the definitions given there are general enough to describe Dactl rewrites as well, at least for part (a). In fact, the semantics thus obtained is the "overwriting" of the final version of Dactl, rather than the "redirection" previously used in earlier versions. This may be seen as confirming the decision to make this change.

We have not yet addressed the question of a more mathematical semantics for control markings. One approach is to consider them as function symbols in their own right. However, this might result merely in an intractable encoding of their role, a more direct treatment being preferable.

6 Conclusion.

We have described a practical language of graph rewriting, and given a wide range of examples of its use. These range from graph manipulation algorithms to translations from functional and logic languages. The semantics of an individual Dactl rewrite agrees with that obtained from the categorical constructions of [Ken90b][1]. Both the design and implementations of Dactl are reasonably stable.

7 Acknowledgements.

Nic Holt, Mike Reeve and Ian Watson made major contributions to the design of Dactl. Kevin Hammond designed the Unix interface. The implementation work was done mainly by Geoff Somner more recently by Ian King. Much of the early work on Dactl was supported by SERC grant no. GR/D59502. Ian King contributed with helpful comments on early drafts of this paper.

---

[1]This correspondence holds only for the final version of the Dactl design, which was significantly influenced by the theoretical work.

8 References.

[Bar87] Barendregt, H.P., van Eekelen, M.C.J.D., Glauert, J.R.W., Kennaway, J.R., Plasmeijer, M.J., and Sleep, M.R., 1987, “Term graph rewriting”, Proc. PARLE conference, Lecture Notes in Computer Science, 259, 141–158, Springer.

[Bar89] Barendregt, H.P., van Eekelen, M.C.J.D., Glauert, J.R.W., Kennaway, J.R., Plasmeijer, M.J., and Sleep, M.R., 1989, “Lean: An Intermediate Language Based on Graph Rewriting” Parallel Computing, 9, 163-177.

[BEG87] Baeten J.C.M, Bergstra J.A. and Klop J.W., "Term Rewriting Systems with Priorities", Rewriting Techniques and Applications, Bordeaux France 1987, Lecture Notes in Computer Science, 257, 83–94, Springer

[Der89] Dershowitz, N., and Jouannaud, J.P., 1989, “Rewrite Systems”, Chap. 15. in Handbook of Theoretical Computer Science, B, North-Holland.

[Ehr79] Ehrig H, "Tutorial introduction to the algebraic approach of graph grammars", in Lecture Notes in Computer Science, 73, 83–94, Springer

[Ehr89] Ehrig, H., and Löwe, M., (eds), 1989, “GRA GRA: Computing by Graph Transformation”, report of ESPRIT Basic Research Action Working Group 3299.

[Far90] Farmer, W.M., Ramsdell, J.D., and Watro, R.J., 1990 “A correctness proof for combinator reduction with cycles”, ACM TOPLAS, 12, 123-134

[Gla88a] Glauert, J.R.W., Hammond, K., Kennaway, J.R., and Papadopoulos, G.A., 1988, “Using Dactl to Implement Declarative Languages”, Proc. CONPAR 88.

[Gla88b] Glauert, J.R.W., and Papadopoulos, G.A., 1988, “A Parallel Implementation of GHC”, Proc. International Conference on Fifth Generation Computer Systems 1988. ICOT, Tokyo.

[Gla88c] Glauert, J.R.W., Kennaway, J.R., Sleep, M.R., and Somner, G.W., 1988, “Final Specification of Dactl”, Report SYS-C88-11, School of Information Systems, University of East Anglia, Norwich, U.K.

[Gre87] Gregory, S., 1987, “Parallel Logic Programming in PARLOG – The Language and its Implementation”, Addison-Wesley, London.

[Ham88] Hammond, K., and Papadopoulos, G.A., 1988, “Parallel Implementations of Declarative Languages based on Graph Rewriting” UK IT 88 Conference Publication, IEE.

[HO82] Hoffmann C. and O'Donnell M.J., 1982, "Programming with equations", ACM Transactions on Programming Languages and Systems, 83-112.

[Hof88] Hoffmann, B., and Plump, D., 1988, “Jungle Evaluation for Efficient Term Rewriting”, Proc. Joint Workshop on Algebraic and Logic Programming, Mathematical Research, 49, 191-203, Akademie-Verlag, Berlin.

[Ken85] Kennaway,J.R. and Sleep,M.R. Syntax and informal semantics of DyNe. in The Analysis of Concurrent Systems, LNCS207, Springer-Verlag 1985.

[Ken87] Kennaway, J.R., 1987, “On ‘On graph rewritings’”, Theor. Comp. Sci., 52, 37–58

[Ken88] Kennaway, J.R., 1988, “The correctness of an implementation of functional Dactl by parallel rewriting”, Proc. Alvey Technical Conference.

[Ken90a] Kennaway, J.R., 1990, “Implementing Term Rewrite Languages in Dactl”, Theor. Comp. Sci., 72, 225-250.

[Ken90b] Kennaway, J.R., 1990, “Graph rewriting in a category of partial morphisms”, paper presented at the Fourth International Workshop on Graph Grammars, Bremen, 1990.

[Ken90c] Kennaway, J.R., 1990, “The specificity rule for lazy pattern-matching in ambiguous term rewrite systems”, Third European Symposium on Programming, LNCS v.432, pp 256–270, Springer-Verlag.

[Klo90] Klop, J.W., 1990, “Term rewriting systems”, Chap. 6. in Handbook of Logic in Computer Science, 1, (eds. Abramsky, S., Gabbay, D., and Maibaum, T.), Oxford University Press.

[Laf89] Lafont Y, "Interaction Nets", LIENS report, Paris 1989 (also 1990 POPL).

[McB87] McBurney, D.L., and Sleep, M.R., 1987, “Transputer-based experiments with the ZAPP architecture”, Proc. PARLE conference, Lecture Notes in Computer Science, 258, 242–259, Springer.

[Pap89] Papadopoulos, G.A., 1989, “Parallel Implementation of Concurrent Logic Languages Using Graph Rewriting Techniques”, Ph.D. Thesis, University of East Anglia, UK.

[Rao84] Raoult, J.C., 1984, “On graph rewritings”, Theor. Comp. Sci., 32, 1–24.

[Ros72] Rosenfeld A and Milgram D.L., "Web automata and web grammars", Machine Intelligence 7 (1972), 307-324.

[Sha86] Shapiro, E.Y., 1986, “Concurrent Prolog: A Progress Report”, Fundamentals of Artificial Intelligence - An Advanced Course, Lecture Notes in Computer Science, 232, Springer.

[Sta80a] Staples, J., 1980, “Computation on graph-like expressions”, Theor. Comp. Sci., 10, 171-185.

[Sta80b] Staples, J., 1980, “Optimal evaluations of graph-like expressions”, Theor. Comp. Sci., 10, 297-316.

[Wad71] Wadsworth, C.P., 1971, Semantics and pragmatics of the lambda-calculus, Ph.D. thesis, University of Oxford.

[Wat87] Watson, I, Sargeant, J., Watson, P., and Woods, V., 1987, “Flagship computational models and machine architecture”, ICL Technical Journal, 5, 555–594.