Journal of Programming Languages **5**, 1997, 85–108

Dactl: an experimental graph rewriting language

John R. W. Glauert[1], Richard Kennaway[1], George A. Papadopoulos[2] and Ronan Sleep[1]

[1]*School of Information Systems, University of East Anglia, Norwich NR4 7TJ, UK*
{jrwg,jrk,mrs}@sys.uea.ac.uk
[2]*Department of Computer Science, University of Cyprus, 75 Kallipoleos Street, POB 537, CY-1678, Nicosia, Cyprus*
george@turing.cs.ucy.ac.cy

A generalized computational model based on graph rewriting is presented along with Dactl, an associated compiler target (intermediate) language. An illustration of the capability of graph rewriting to model a variety of computational formalisms is presented by showing how some examples written originally in a number of languages can be described as graph rewriting transformations using Dactl notation. This is followed by a formal presentation of the Dactl model before giving a formal definition of the syntax and semantics of the language. Some implementation issues are also discussed.

Keywords: term graph rewriting, functional and logic programming, intermediate (compiler target) languages, language embeddings

Modern programming languages make heavy use of complex data structures to represent lists of objects or tree structures. A common representation of these structures is in terms of records containing pointers to other records. In mathematical terms, such structures are ordered directed graphs, or hypergraphs (Hoffman and Plump 1988).

These data structures are copied and modified during the course of evaluation. In mathematical terms, evaluation may be seen as a process of transforming graphs according to rules determined by the program text. Hence a program may be seen as specifying a graph rewriting system.

In functional languages, graphs are commonly used to represent data values. In lazy languages, closures for unevaluated expressions will also take the form of graph structures. Graph rewriting is accepted as an efficient technique for implementing lazy functional languages (Peyton Jones 1987).

In logic languages graph structures can be used to represent terms. Terms may contain uninstantiated variables which are shared between terms in the program goal. The list of goals itself is naturally represented as a graph and the process of resolution becomes a process of transforming the goal graph in search of a solution. The effectiveness of using graph rewriting in the implementation of concurrent logic languages in particular is demonstrated in Papadopoulos (1989).

Dactl is a very general model of computation based on graph rewriting. Using an interpreter for the notation, it has been possible to demonstrate working compilers for a surprisingly wide range of languages

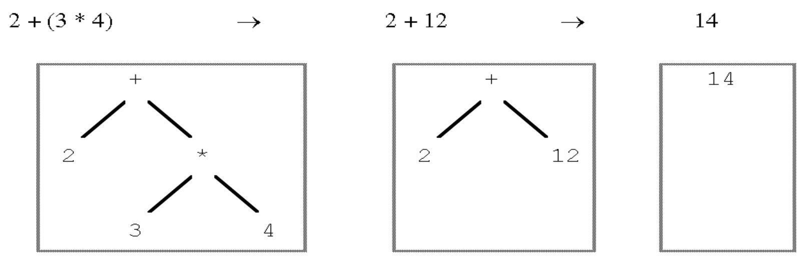


Figure 1: Rewriting of terms

including Hope, LISP, Parlog, GHC, ML and Clean (Glauert et al. 1988a, Glauert and Papadopoulos 1988, Hammond and Papadopoulos 1988, Hammond 1990, Kennaway 1990a).

The underlying technique of graph rewriting is common to all these experimental implementations, but conventions governing the representation of data and the scheduling of computation differ. However, more recent work is focusing on the possibility of uniting the implementations of a number of languages at the implementation level. Such work looks at combining logic and functional styles (Papadopoulos 1997a) and integrating functional and concurrent programming (Glauert 1992, Glauert et al. 1993).

An important benefit of using a rewriting model of computation is that often the order in which rewritings take place does not affect the final result. This is the **confluence** or **Church–Rosser** property. Further, when a number of independent rewrites may be made it is often possible for them to be performed concurrently. Dactl supports concurrent evaluation, and makes explicit any opportunities for parallel execution.

This paper gives some examples of how computation in a number of languages may be described as graph rewriting, giving the Dactl notation for the examples shown. It goes on to present the Dactl model more formally before giving a formal definition of the syntax and semantics of the language.

1 Examples of computation by graph rewriting

In this section we use the Dactl notation to give a textual representation of graph rewriting rules used to transform some program graphs. Pictorial representations are also shown. The reader should not be unduly concerned with details of the notation on first reading.

1.1 Evaluation of expressions

It is a familiar notion to regard the reduction of expressions as rewriting of terms or trees, which are restricted forms of graphs (Fig. 1). The usefulness of graph representation is revealed if we have a call to a function square which is evaluated lazily (Fig. 2).

In the Dactl representation of these computations, even primitive arithmetic operations are represented as functions. The programs are given in Fig. 3.

The symbols *, # and ^ are used to control the order of evaluation, as explained in detail below. For now it suffices to say that the symbol * is a **spark** indicating the next expression to be reduced. It is necessary to make the control of evaluation explicit in Dactl since the notation will be used to model computations requiring a range of different evaluation strategies.

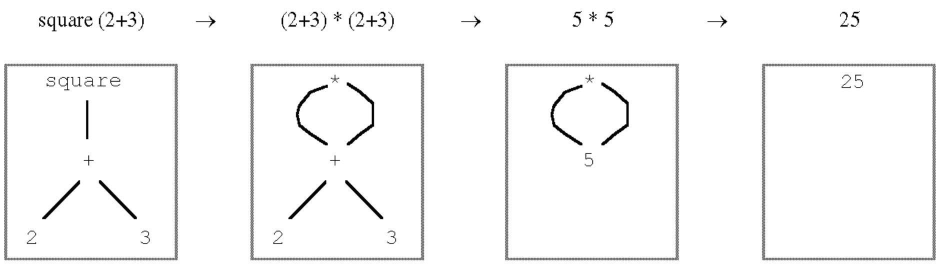


Figure 2: Lazy evaluation

```
MODULE ExprEx;
IMPORTS Arithmetic;
RULE
INITIAL ⟹ #IAdd[2 ^∗IMul[3 4]];
ENDMODULE ExprEx;

MODULE SquareEx;
IMPORTS Arithmetic;
SYMBOL REWRITABLE Square;
RULE
Square[n] ⟹ #IMul[^∗n n];
INITIAL ⟹ ∗Square[IAdd[2 3]];
ENDMODULE SquareEx;
```

Figure 3: *ExprEx* and *SquareEx* modules

$fun\ NFib\ 0 = 1$
$|\quad NFib\ 1 = 1$
$|\quad NFib\ n = NFib(n-1) + NFib(n-2) + 1$

$MODULE\ NFib;$
$IMPORTS\ Arithmetic;$
$SYMBOL\ REWRITABLE\ PUBLIC\ CREATABLE\ NFib;$

$RULE$
$NFib[(0+1)] \implies *1;$
$NFib[n] \implies \#\#IAdd[\hat{}\#NFib[\hat{}*ISub[n\ 1]]\hat{}\#IAdd[\hat{}\#NFib[\hat{}*ISub[n\ 2]]\ 1]];$
$ENDMODULE\ NFib;$

Figure 4: *NFib* function

The sequence of evaluation in these cases is as follows:

$$*INITIAL \to \#IAdd[2\ \hat{}*IMul[3\ 4]] \to \#IAdd[2\ \hat{}\ *12] \to *IAdd[2\ 12] \to *14$$

$$*INITIAL \to *Square[IAdd[2\ 3]] \to \#IMul[\hat{}\ *n : IAdd[2\ 3]\ n] \to \#IMul[\hat{}\ *n : 5\ n] \to IMul[n : 5\ n] \to *25$$

Note the use of the identifier n to indicate the sharing of the sub-expression $IAdd[2\ 3]$.

1.2 Exploiting parallelism

Here we define the much used *NFib* function which counts function calls. In Fig. 4 the function is given first in Standard ML and then in Dactl. The translation is for a strict language (such as Standard ML) and evaluation of the two recursive calls to *NFib* can proceed in parallel. In fact, to be faithful to the semantics of SML, we must check that evaluation would have no side-effect and that no exceptions could be raised.

An interesting point here is the definition of the *NFib* symbol in Dactl. Its **access class** is rewritable within the module that contains the rules that rewrite *NFib* nodes, but only creatable when exported. Thus importing modules can create active *NFib* nodes, but cannot define any new rules for the symbol.

1.3 Programming with state

The rewriting shown in Fig. 4 repeatedly replaces a node representing some expression with a new expression with the same extensional value. Ultimately, a base value such as an integer is produced.

When programming with state, a more flexible style of rewriting is needed to express values which can have mutable values. The language SML uses the concept of reference values which capture the concept in a way which can be modelled by graph rewriting. A function *ref* creates reference values with an initial

```
MODULE RefEx;
IMPORTS Arithmetic;

SYMBOL CREATABLE Unit;
SYMBOL OVERWRITABLE Ref;
SYMBOL REWRITABLE Assign; DeRef; Seq; LetRes;

RULE
INITIAL ⟹ #Seq[^*Assign[r 6] LetRes[r]], r : Ref[4];
LetRes[r] ⟹ #IAdd[^*DeRef[r] 2];
Assign[r : Ref[o] n] ⟹ *Unit, r := Ref[n];
DeRef[Ref[v]] ⟹ *v;
Seq[Unit b] ⟹ *b;
ENDMODULE RefEx;
```

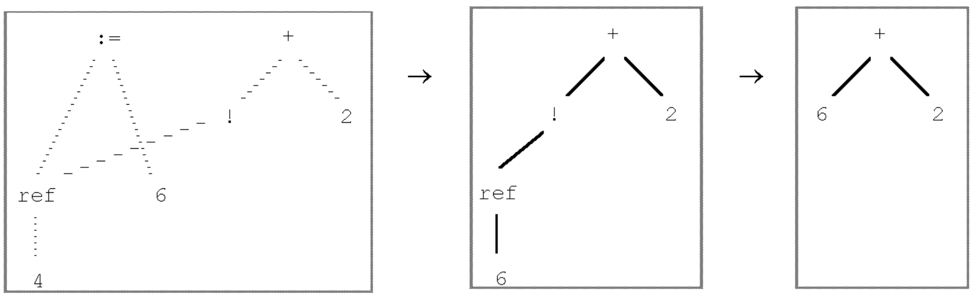


Figure 5: Dactl code for *RefEx* module

value; the operator := is a function which assigns a new value to a reference and returns the unit value. The operator ! is used for de-referencing.

We model reference values as nodes whose contents can change during evaluation. Consider the expression:

let val $r = ref\ (4)$
in $r := 6;\ !r + 2$
end

Sequencing of evaluation ensures that *r* holds the value 6 by the time the value is de-referenced. Dactl code for this program might be as shown in Fig. 5. The novel feature is in the rule for assignment. The *Assign* node is rewritten to a unit value, but at the same time, the first argument, a *Ref* node, is overwritten with a new *Ref* node with different contents. Other parts of the graph with pointers to this node will now find a different value if they apply the de-referencing operation to the node.

LetRes represents the computation required after the first statement of the sequence has completed.

1.4 Logic programming

Variables in conventional languages correspond to the references of SML considered above. Variables in logic programming languages have a very different meaning, but similar graph rewriting techniques may be used.

In this paper we highlight the particular class of concurrent logic programming languages (Shapiro 1989). Here goals may be viewed as processes, with variables acting as synchronizing communication channels between the goal processes. In a Dactl translation, goals become rewritable graph nodes, and terms become graph structures including mutable nodes representing variables.

Dactl control markings are used to synchronize computation. If a goal can neither succeed nor fail until a variable has been instantiated, the computation suspends waiting for the variable to be given a value by output unification in some other goal. When this takes place, the original computation is reactivated and will be able to make further progress.

The example below, the **unavoidable append program**, serves to illustrate some of the techniques used in modelling a concurrent logic program as a set of graph rewriting rules (for a full treatment of the relationship between concurrent logic programming and graph rewriting see Papadopoulos (1997a)). The example has been written in a kernel-like form making explicit the input and output unification:

$$append([H|T],Y,Z) : -true|Z = [H|Z1], append(T,Y,Z1).$$
$$append([\,],Y,Z) : -true \mid Z = Y.$$

$$? - append(P,Q,Ans),\ P = [1],\ Q = [2].$$

Dactl code for the same example might be as in Fig. 6. Input unification is a pattern-matching process and will not instantiate variables in the goal. Hence if the first argument to $Append$ is a variable, the goal is suspended, waiting for the variable to be instantiated. This is modelled by the third rule in the Dactl code for $Append$. The process will be reactivated when unification succeeds. Note also the way $Unify$ is implemented by means of non-root overwrites.

In general, a procedure written in a concurrent logic language and comprising n clauses is translated into a set of $n+2$ Dactl rules: one for each clause, one for modelling suspension and one for reporting failure. In addition, if a body clause comprises m goals ($m > 1$), it is translated into an equivalent set of active Dactl nodes that execute concurrently as arguments of an m-argument monitoring *And* function that will rewrite to *Fail* as soon as any of its arguments reports failure and to *Succ* otherwise. In many cases, however, similar input unifications at the source level can be collapsed into a single operation at the Dactl level, thus reducing the number of generated rules, and producing more efficient code. The extra machinery for reporting on failure is not needed for those concurrent logic languages whose semantics do not reflect on failure. Assuming Strand or Janus semantics (see Papadopoulos (1997b)) the equivalent Dactl code for the above program is shown in Fig. 7. Note that since Strand and Janus support assignment rather than full output unification, a simple redirection of the overwritable node *Var* suffices to implement the instantiation of variables without the need for a set of $Unify$ rules. In these languages the state of the computation is undefined if during the assignment the variable involved in the operation turns out to be already instantiated to some value. This behaviour, which is reminiscent more of functional languages

```
MODULE Append;

SYMBOL CREATABLE Ans; Fail; Succ; Cons; Nil;
SYMBOL REWRITABLE Unify; Append; And;
SYMBOL OVERWRITABLE Var;

RULE
INITIAL ⟹ Ans[ans : Var],
   *Append[p q ans],
   *Unify[p : Var Cons[1 Nil]], *Unify[q : Var Cons[2 Nil]];

Append[Cons[h t] y z] ⟹
   #And[^*Unify[z Cons[h z1 : Var]] ^ *Append[t y z1]];
Append[Nil y z] ⟹ *Unify[z y];
Append[l : Var y z] ⟹ #Append[^l y z];
Append[ANY ANY ANY] ⟹ *Fail;

Unify[v : Var term] ⟹ *Succ, v := *term;
Unify[Cons[h1 t1]Cons[h2 t2]] ⟹
   #And[^*Unify[h1 h2] ^ *Unify[t1 t2]];

And[Succ Succ] ⟹ *Succ;
r : And[p1 : (ANY − Fail) p2 : (ANY − Fail)] → #r;
And[ANY ANY] ⟹ *Fail;
ENDMODULE Append;
```

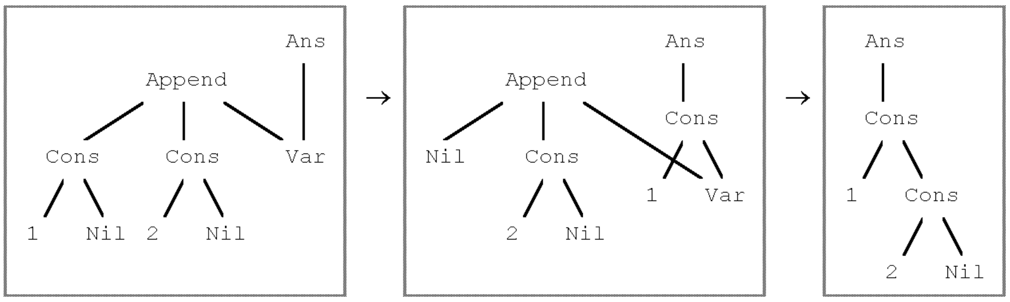


Figure 6: Unavoidable append program

$MODULE\ Append;$

$SYMBOL\ CREATABLE\ Ans;\ Cons;\ Nil;$
$SYMBOL\ REWRITABLE\ Append;$
$SYMBOL\ OVERWRITABLE\ Var;$

$RULE$
$INITIAL \implies Ans[ans:Var],$
$\quad *Append[p\ q\ ans],\ p:Cons[1\ Nil],\ q:Cons[2\ Nil].$

$Append[Cons[h\ t]\ y\ z:Var] \to z := *Cons[h\ z1:Var],\ *Append[t\ y\ z1];$
$Append[Nil\ y\ z:Var] \to z := *y;$
$Append[l:Var\ y\ z] \to \#Append[^\wedge l\ y\ z];$

$ENDMODULE\ Append;$

Figure 7: Equivalent Dactl code

rather than (concurrent) logic ones, allows more efficient implementations at the expense of somewhat compromising the languages semantics. More information on how languages like Strand and Janus would be mapped on a CTL like Dactl can be found in Papadopoulos (1989), Papadopoulos (1997a).

1.5 Concurrent programming

In the last example, logic variables could be used for communication. Graph rewriting may be used to model computation in concurrent programming languages where there are explicit notions of processes and communication channels.

We give an example in a possible translation of Facile (Giacalone et al. 1989), a language which integrates concurrent and functional programming styles in a symmetric fashion. The language has been implemented as an extension to SML. In the following fragment a value is communicated over a channel named c which links two processes executing in parallel:

$let\ val\ c = Chan$
$in \ldots c!4 \ldots \| \ldots 3 + c? \ldots$
end

The operator ! is here used to indicate the sending of a value over the channel given as the first argument. The result is the unit value. ? indicate reception of a value which is returned as the result. Communication is synchronous, so neither process continues until the communication has been closed (see Fig. 8). As in the logic example, the *Var* nodes act as place-holders for values not yet available. If the reader of the channel executes first, the synchronizing variable is stacked. When the writer executes, the input queue in the channel identifies the appropriate reader; its variable is overwritten and a unit value is returned. If

```
MODULE CommEx;
IMPORTS Arithmetic;

SYMBOL CREATABLE IQ; OQ; Nil; Unit;
SYMBOL OVERWRITABLE Chan; Var;
SYMBOL REWRITABLE Get; Put;

RULE
Get[c : Chan[Nil iq]] ⟹ r : Var, c := Chan[Nil IQ[r iq]];
Get[c : Chan[OQ[v : INT s : Var oq] iq]] ⟹ *v, s := *Unit,
   c := Chan[oq iq];

Put[c : Chan[oq Nil] v] ⟹ s : Var, c := Chan[OQ[v s oq]Nil];
Put[c : Chan[oq IQ[r : Var iq]]v : INT] ⟹ *Unit, r := *v,
   c := Chan[oq iq];

INITIAL ⟹ #IAdd[3 ^ * Get[c]], *Put[c 4], c : Chan[Nil Nil];
ENDMODULE CommEx;
```

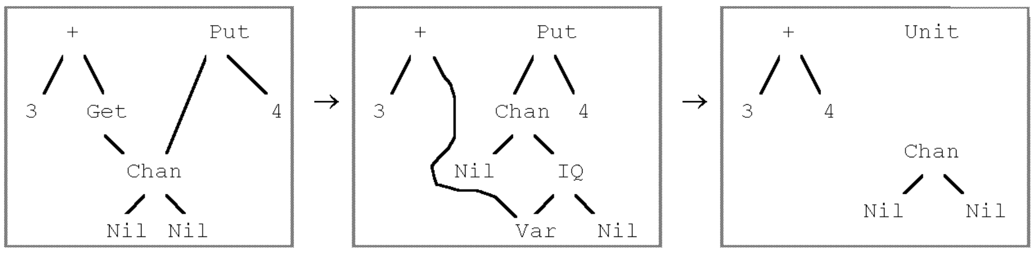


Figure 8: *CommEx* module

the writer executes first, both the value and the synchronizing variable are stacked. If multiple readers precede a writer, then all are stacked, and vice versa.

In this section we have shown that graph rewriting may be used to model the evaluation of core constructs of languages representing a large range of programming styles. Code in the practical graph rewriting language Dactl has been given. This code is executable using the reference interpreter written for the language by the authors and their associates. Some more information about the implementation of Dactl is given in section 2.1, after a detailed and formal description of the language.

2 A formal introduction to Dactl

2.1 Dactl in context

Term (or **tree**) rewriting systems have proven useful both as specifications and occasionally as practical systems for symbolic computation (see Hoffman and O'Donnell (1982) for a practical system with a sound theoretical underpinning). Klop (1990) and Dershowitz and Jouannaud (1990) 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 Rosenfeld and Milgram (1972), and a significant body of theory has been developed, most notably by the Berlin school of Ehrig and others: Ehrig and Löwe (1989) gives an authoritative overview.

Practical uses of graph rewriting date back at least to Wadsworth (1971), 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 (Staples 1980a, Staples 1980b, Barendregt et al. 1987, Hoffman and Plump 1988, Farmer et al. 1990). The main result is that sharing implementations produce the correct semantics at least for **orthogonal** term rewrite systems.

Logic programming languages of the committed choice variety (for example Concurrent Prolog (Shapiro 1989) and Parlog (Gregory 1987)) may be viewed as specialized graph rewriting languages, as may actor models such as DyNe (Kennaway and Sleep 1983). More recently Lafont (1990) has proposed an Interaction Net model of computation which again may be viewed as specialized graph rewriting, whose constraints are inspired by Girard's work on linear logic. The precise relationship between interaction nets and term graph rewriting systems as expressed by languages like Dactl is studied in Banach and Papadopoulos (1997).

In 1983 three of the present 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 compiler 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. For instance, we found no straightforward way to specify normal, sequential computation using built-in operators. Furthermore, the markings and notification mechanism were over-general and made it difficult to understand how computation would proceed in some cases. What was actually needed was a CTL at a level (somewhat) lower than that of its computational model (whereas Dactl was a direct realization of the computational model it advocated) where even more precise control of evaluation order would be possible to specify. 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 (Hammond and Papadopoulos 1988, Glauert et al. 1988a, Glauert and Papadopoulos 1988, Papadopoulos 1989, Hammond 1990, Kennaway 1990a).

The CTL motivations of the Dactl project are now mainly historical. What remains is one of the few genuine graph rewriting language implementations in existence. There is a stable, reasonably engineered interpreted implementation of Dactl under Unix with modular compilation facilities and a comprehensive Unix interface, and a more recent implementation for Macintosh computers which is in regular use. An experimental compiler has been developed for a simplified form of the language. This form, referred to as MONSTR (Banach and Watson 1988, Banach 1993), was particularly suited for distributed machines such as Flagship (Watson et al. 1988). MONSTR placed some mainly syntactic (and thus trivially statically checkable) restrictions on the types of programs that could be written. These restrictions (such as enforcing a maximum of one non-root redirection per rule and disallowing deep pattern matching) reduced significantly the amount of locking that had to be done in order to guarantee adherence to the intended operational semantics of the computational model. However, some liveness constraints were still hard to check. 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.

2.2 Dactl graphs

We present the syntax for Dactl graphs using rules from the full syntax which appears in the Appendix.

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. The syntax allows for standard representations of values for numbers, characters, and strings, etc. The user may introduce named symbols using identifiers starting in upper case:

$$\begin{array}{ll}
\mathit{Item} & ::= \mathit{SYMBOL\ AccessClass}[\mathit{PUBLIC\ AccessClass}]\ \mathit{SymbolList} \\
 & \quad \mid \ldots \\
\mathit{AccessClass} & ::= \mathit{READABLE} \mid \mathit{CREATABLE} \mid \mathit{OVERWRITABLE} \mid \mathit{REWRITABLE}\ . \\
\mathit{SymbolList} & ::= \mathit{Symbol}; \{\mathit{Symbol};\}\ . \\
\mathit{Symbol} & ::= \mathit{Upper}\{\mathit{IdentChar}\}\ . \\
\mathit{IdentChar} & ::= \mathit{Upper} \mid \mathit{Lower} \mid \mathit{Digit} \mid . \mid \_ \mid ' \mid ''.
\end{array}$$

Some predefined symbols are available, such as *Cons* and *Nil* for list construction, and *True* and *False* for Booleans. The concept of **access class** defines the role of a symbol: creatable symbols are constructors; overwritable symbols are mutable values; and rewritable symbols generally name functions. The classes control where symbols may appear in rule patterns and bodies. A primitive module facility, discussed fully in Glauert et al. (1988b), restricts the usage of symbols when imported as indicated by the *PUBLIC* access class.

From nodes 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**. The successors may be indicated by a node identifier or **nodeid**, or by a nested node definition. The presentation below omits node and arc markings which are used to control evaluation and are introduced later:

$$\begin{array}{ll} Graph & ::= NodeDefinition\{,NodeDefinition\}\ . \\ NodeDefinition & ::= [Nodeid\ :]Node\ . \\ Node & ::= DataValue \mid Symbol[[Term\{Term\}]]\ . \\ Term & ::= Nodeid \mid NodeDefinition\ . \\ Nodeid & ::= Lower\{IdentChar\}. \end{array}$$

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. Note that until the final form is computed, some nodes unreachable from the root may influence the final form.

The following examples give the textual representation of two graphs. The first is an example of a DAG and the second an example of a cyclic graph.

Example 1 A DAG

$$\begin{array}{l} r : Append[s\ s], \\ s : Cons[z\ n], \\ z : 0,\ n : Nil \end{array}$$

Example 2 A cyclic graph

$$\begin{array}{l} c : Cons[o\ c], \\ o : 1 \end{array}$$

Both examples require the use of node identifiers as there is sharing of arguments. Alternative graphs with the same meanings are $Append[s : Cons[0\ Nil]\ s]$ and $c : Cons[1\ c]$. Terms with a tree structure can be represented without using node identifiers.

2.3 Dactl rewriting rules

The reduction relation for a Dactl system is described by a set of rewriting rules which describe **graph transformations**. A set of **control markings** is used to specify when the transformations may be applied.

The left-hand side of a rule consists of a **pattern** which is a generalization 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 & representing **union**, **difference** and **intersection** respectively. The generalized form of node specification is given in Fig. 9. 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 of the pattern and the graph undergoing rewriting.

$NodeDefinition$	$:= [Nodeid :]\ Node\ .$
$Node$	$:= DataValue \mid Symbol[[Term\{Term\}]]$
	$\mid Term\{PatternOpTerm\})$
	$\mid SymbolClass$
$SymbolClass$	$::= DataClass$
	$\mid ANY$
	$\mid AccessClass.$
$DataClass$	$::= INT \mid LONG \mid REAL \mid BOOL \mid CHAR \mid$
	$\mid STRING \mid PTR \mid VECTORC \mid VECTORO\ .$
$PatternOp$	$::= + \mid\mid \&.$

Figure 9: Generalized form of node specification

The right-hand side of a rule includes the **contractum graph**, a number of **redirections**, and a number of **activations**. 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. The control markings, including the **activations**, are used to identify redexes for future rewrites. Very frequently only a single redirection of the root is intended, and the syntax of Dactl provides a special connective $\Longrightarrow$ between the left- and right-hand side of a Dactl rule for this purpose:

$Rule$	$::= Pattern \Longrightarrow [ContractumPart\{,ContractumPart\}]$
	$\mid Pattern \Longrightarrow Term\{,ContractumPart\}.$
$ContractumPart$	$::= NodeDefinition \mid Activation \mid Redirection\ .$
$Activation$	$::= Nodeid\ .$
$Redirection$	$::= Nodeid := Term\ .$

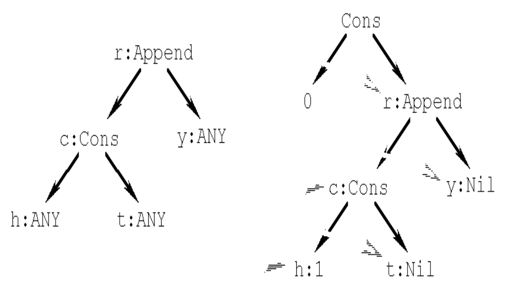
The following example rules model the appending of lists:

$$Append[Cons[h\,t]\,y] \Longrightarrow Cons[h\,Append[t\,y]]|$$
$$Append[Nil\,y] \Longrightarrow y;$$

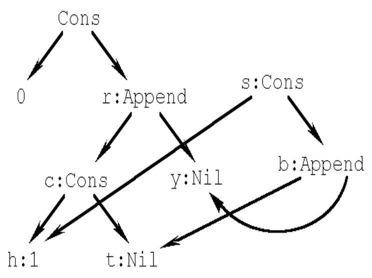
Dactl rules which do not use the connective symbol $\Longrightarrow$ use the connective symbol $\rightarrow$. This symbol has no semantic connotations and serves merely as a delimiter for the two sides of the rule. Its use means that all redirections have to be explicitly stated. The following example shows the append rules using the $\rightarrow$ connective and the symbol class ANY:

$$r : Append[Cons[h : ANY\ t : ANY]\ y : ANY] \rightarrow s : Cons[h\,Append[t\,y]], r := s|$$
$$r : Append[Nil\,y : ANY] \rightarrow r := y;$$

The use of priority rewrite semantics is both common and convenient, and so Dactl supports it. Rules separated by a semicolon are matched in order, whereas rules separated by a | may be dealt with in any order.



(a) Matching



(b) Building

Figure 10: Stages of rewriting

Constraints are imposed on where symbols of particular access classes may appear: **rewritables** may only appear as the root of patterns; **creatables** and **overwritables** only below the root; redirection of nodes below the root is only permitted if they are **overwritables**.

The syntax for a complete module including a set of rules is:

Module ::= *MODULE Symbol*; $\{$*Item*$\}$*ENDMODULE Symbol*; .
Item ::= *RULE RuleList*
| . . . .
RuleList ::= *RuleGroup*; $\{$*RuleGroup*; $\}$.
RuleGroup::= *Rule*$\{|$ *Rule*$\}$.

2.4 Graph rewriting in Dactl

We will describe Dactl rewriting in more detail. The four stages of rewriting are **matching**, **building**, **redirection** and **reactivation**.

2.4.1 Matching

A match is a graph homomorphism from (the nodeids of) the pattern of a rule to (the nodeids of) the program graph. Structure is preserved by this mapping, except at variable nodes (marked by a symbol class such as *ANY*). As an example consider the diagram in Fig. 10a. It shows the matching of the pattern of the first append rule to a piece of graph. The horizontal arrows show the homomorphic relationship between the two graphs. In this example, the redex is the graph rooted at the *Append* node. The matching process also constructs a binding between nodes in the pattern and the program graph which will used in the later phases.

Pattern nodes whose symbols are symbol class names are matched specially: *ANY* matches unconditionally and data classes match values of the appropriate type. More complex matching rules apply when

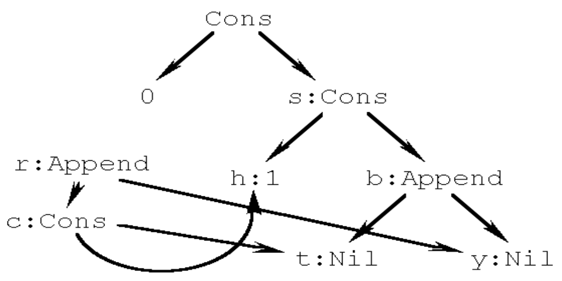


Figure 11: Redirection

pattern operators are used, as explained in Glauert et al. (1988b).

2.4.2 Building

The second phase of rewriting builds a copy of the contractum of the matched rule. The contractum may contain occurrences of identifiers from the pattern. During building, such occurrences become arcs to the corresponding nodes matched in the first phase. The building phase is not required for selector rules, such as the second *Append* rule, which have no contractum. Figure 10b shows our example program graph once the contractum of the matched append rule has been constructed.

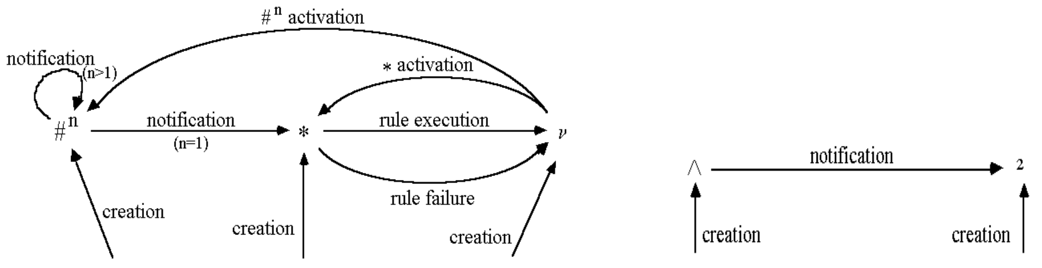
2.4.3 Redirection

The next stage of a rewrite is redirection. Its purpose 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 nodeids identified by the pattern variables may be redirected within a single atomic rewrite. Our running example only has a single redirection and redirects references to the node which matches the root of the pattern to the *Cons* node constructed during the building phase. Figure 11 shows the state after the redirection. Note how all references have been redirected and thus the old *Append* node has been disconnected from the root of the program graph. This is now garbage in the Dactl sense and a common implementation technique is to overwrite it with the contents of *s*, thus avoiding the overhead of supporting genuine physical redirection of pointers.

Certain constraints on Dactl rules are necessary to avoid inconsistent redirections (Glauert et al. 1988b). In particular, the sources of all redirections must be distinct which means in general that they must have explicit and disjoint symbols.

2.4.4 Activation

The final phase of rewriting is activation, which allows the alteration of the control state of pre-existing nodes in the program graph. The control states of nodes are used to spark or suspend parts of the program graph as described in the next section. Control markings have been omitted in our example above.



(a) Node transitions (b) Arc transitions

Figure 12: State transitions of nodes and arcs

2.5 Control of evaluation in Dactl

Dactl is concerned with control of evaluation as well as the properties of an abstract rewriting system. It includes fine grain control and synchronization markings as an integral part. These markings are attached to particular nodes and arcs in the program graph. The intuitive basis of the markings is that there may be multiple control loci each of which is a node where reduction may take place.

2.5.1 Dactl markings

The full syntax includes provision for markings on symbols and nodeids:

```
Node       ::= [NodeMark] Symbol[[MarkedTerm{MarkedTerm}]]|.
NodeMark   ::= * | #{#} .
MarkedTerm ::= [ArcMark] Term .
ArcMark    ::= ^ .
Activation ::= [NodeMark] Nodeid .
```

The most important **node** marking is the **activation** denoted by *. Only activated nodes are considered as the starting points of rule matching. A node may be created active, or may become active during the **activation** phase of rewriting.

The other node 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 synchronization between suspensions and notifications involves arc markings.

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 at the target 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.

The possible state transitions of nodes and arcs are summarized in Fig. 12. Operations having no effect (e.g. activation of non-idle nodes) are omitted.

The full syntax in the Appendix specifies the places where markings may appear; however, the use of all of these markings is illustrated by our append rules below. These are the same rules as before but now contain the relevant markings. The first rule suspends the *Cons* node until the append of the rest of the list is complete and the notification removes the suspension. The activation in the second rule fires the list y thus starting the notification process back up the spine of any constructed list:

$$Append[Cons[h\,t]\;y] \implies \#Cons[h\,{}^{\wedge}*Append[t\;y]]\,|$$
$$Append[Nil\;y] \implies *y$$

When rewriting with markings, new graph nodes are built with markings as specified in the contractum. Before redirection, any markings on the original node are removed before arcs are redirected to the new node. Finally, the activation phase may mark other nodes matched by the pattern.

2.5.2 Dactl rewriting with markings

We have discussed the application of a single Dactl rule. Execution of a program will involve the application of many rules. In each case the application of a rule takes place only at an activated node.

When a node is activated it may match a rule and be rewritten. However, the node may also be a data value such as an integer, or a user-defined constructor value. In such cases, no rules apply and the activation is discarded. In this circumstance it is likely that some parent computation may be able to make progress. So notification occurs for each notification path which leads to the chosen node as described in the last section.

Each program must include a rule whose left-hand side has the pattern $INITIAL$. The initial Dactl graph contains an active node which will match this rule. As a result of rewriting, new activations will be generated. If there are several active nodes, they may be processed in any order, or even concurrently as long as the effect is the same as could have been achieved by chosing active nodes one at a time. Once a Dactl program graph contains no activations, execution is complete and the graph viewed from the root is the result of computation.

The following example shows how the markings upon the append rules work in practice. Take the following graph:

$$a : *Append[k : Cons[o : 1\; n : Nil]\; k]$$

The subgraph rooted at the active node matches the first rule and the reduction produces the following graph.

$$m : \#Cons[o : 1^{\wedge}b : *Append[n : Nil\; k : Cons[o\; n]]],$$
$$a : Append[k\; k]$$

It can be seen that the matching matched the node k to two different parts of the pattern. In Dactl, it is perfectly consistent for a tree-structures pattern to match a graph with sharing. The original node a is now garbage, because it is disconnected from the root node m, and so can be removed. The new active *Append* node now matches the second rule and the following graph is produced:

$$m : \#Cons[o : 1\;{}^{\wedge}k : *Cons[o\; n : Nil]],$$
$$b : Append[n\; k]$$

The node *b* is now garbage and again can be removed. The *Cons* node is active but is a constructor and so does not match any rules. This is a match failure and thus the notification is removed and the suspension marking upon *m* is changed into an activation marking:

$$m : {*}Cons[o : 1\ k : Cons[o\ n : Nil]]$$

Again the *Cons* node *m* suffers match failure and so the final form is as shown below. Note that eventually the *Append* function causes match failure at the root of the constructed list. This allows consumers of the append operation to suspend awaiting the termination of the function:

$$m : Cons[o : 1\ k : Cons[o\ n : Nil]]$$

2.6 Dactl modules

To facilitate separate compilation, a Dactl program is split into a collection of modules. This provides a way of organizing a Dactl program into meaningful components and controlling their access to each other. Modules may import other modules and this is the way Dactl is interfaced to the real world. A substantial set of interface modules is defined in Glauert et al. (1988b).

Each module contains a set of symbol and rule definitions along with import statements. The symbol definitions give the class of each symbol either *CREATABLE*, *REWRITABLE*, or *OVERWRITABLE* corresponding to constructors, functors and variables respectively. The symbol definitions also state whether a symbol is *PUBLIC* or not and its public class. Rules are defined as shown earlier and they are all imported along with their module.

The following example shows a module containing the append rules. Note how the list module, which contains the definitions of *Cons* and *Nil*, is imported and that although the *Append* symbol is *REWRITABLE* within the module, its *PUBLIC* class is *CREATABLE*. This means that the rules may be used by an importing module which creates active *Append* nodes, but no new rules for *Append* may be added:

$MODULE\ Append;$

$IMPORTS\ Lists;$
$SYMBOL\ REWRITABLE\ PUBLIC\ CREATABLE\ Append;$

$RULE$
$Append[Cons[h\ t]\ y] \implies \#Cons[h\ {}^\wedge {*}Append[t\ y]]\ |$
$Append[Nil\ y] \implies {*}y;$

$ENDMODULE\ Append;$

3 Conclusions

3.1 Dactl implementation

The original aim was for Dactl to act as a compiler target language. This happened to some extent as part of the UK Alvey Flagship project with which the authors were associated. However, the main value of

Dactl has been as a practical notation for expressing and exploring implementation techniques based on graph rewriting. To this end it has been important to have available a reliable reference implementation of the language.

The reference interpreter was designed to be faithful to the model, and to provide extensive profiling and tracing facilities, with efficiency considered as a secondary concern. However, the interpreter has proven an effective tool for exploring some non-trivial programs resulting from experimental compilers for SML (Hammond 1990) and GHC (Papadopoulos 1997a). The performance of the interpreter compared favourably with available implementations of concurrent logic languages during exploration of graph rewriting techniques for implementing the same languages.

In order that Dactl could be used to explore a wide range of graph rewriting techniques, it was decided to make the notation very general. The notation allows some very complex patterns to be provided for rules. This makes efficient pattern-matching code hard to produce. Also, Dactl does not constrain the arity of symbols, requiring additional checks during matching. The patterns of control that can be expressed using the Dactl marking scheme make scheduling of execution hard to predict.

Experience with use of Dactl suggests that quite simple patterns are usually sufficient and that pattern-matching for such rules need not be costly. In particular, there is little to be gained from allowing symbols to have variable arity. By keeping to some simple rules for expressing control of evaluation, it is possible to take advantage of implementation techniques used by compilers for SML: continuation passing style (Appel 1992) and Haskell (the STG machine (Peyton Jones and Salkild 1989). An experimental compiler generating C code for Dactl (with some minor restrictions) proved that suitably constrained variants of Dactl could still provide an effective implementation route.

Recently, we have used the interpreter as a means of exploring and exploiting the relationship between the term graph rewriting systems framework and other models and programming paradigms. For instance, Glauert (1992, Glauert et al. 1993) study the relationship between graph rewriting and process calculi by mapping the process language Facile; Banach and Papadopoulos (1997) provide a mapping between graph rewriting and interaction nets and Papadopoulos (1996, Papadopoulos 1997b) extend the basic term graph rewriting model with object oriented capabilities.

3.2 Summary of main features of the Dactl language

Although it is clear that the design of Dactl could be improved, we believe that there is much to be learned by studying the present design. The current definitive reference document for Dactl is Glauert et al. (1988b), obtainable from the authors. Dactl graphs are **term graphs** in the sense of Barendregt et al. (1987). 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 by Hoffman and Plump (1988).

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 (Kennaway 1988).

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.

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 (1990a), and Hammond and Papadopoulos (1988).

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.

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

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 Kennaway (1990b). Both the design and implementations of Dactl are reasonably stable.

Acknowledgements

Nic Holt, Mike Reeve and Ian Watson made major contributions to the design of Dactl. Kevin Hammond designed the Unix interface. The initial implementation work was done by Geoff Somner and later continued by Ian King who produced an experimental compiler written in SML. Much of the early work on Dactl was supported by SERC grant no. GR/D59502.

Work on the more formal aspects was aided by collaboration with the Dutch Parallel Reduction Machine Project led by Professor H.P. Barendregt. This collaboration led to a number of joint publications (Barendregt et al. 1987, Barendreght et al. 1989), 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).

References

Appel, A.W. (1992) *Compiling with Continuations*. Cambridge University Press, Cambridge.

Banach, R. (1993) MONSTR: term graph rewriting for parallel machines. *In Term Graph Rewriting: Theory and Practice* (eds M.R. Sleep, M.J.Plasmeijer and M.C.J.D.van Eekelen), Wiley, New York, pp. 243–52.

Banach, R. and Papadopoulos, G. A. (1997) A study of two graph rewriting formalisms: Interaction Nets and MONSTR. *Journal of Programming Languages,* to appear.

Banach, R. and Watson, P. (1988) Dealing with state in Flagship: the MONSTR computational model. *In Proc. CONPAR88,* Manchester, UK, 12–16 September, Cambridge University Press, Cambridge, pp. 595–604.

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. *In Proc. PARLE conference*, Lecture Notes in Computer Science **259**, Springer-Verlag, Berlin, pp. 141–58.

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**, 163–77.

Dershowitz, N. and Jouannaud, J.P. (1990) Rewrite systems. Chapter 6 *in Handbook of Theoretical Computer Science* B, North-Holland, Amsterdam, pp. 243–320.

Ehrig, H. and Löwe, M. (eds.) (1989) *Gra Gra: Computing by Graph Transformation*. Report of Esprit Basic Research Action Working Group 3299.

Farmer, W.M., Ramsdell, J.D. and Watro, R.J. (1990) A correctness proof for combinator reduction with cycles. *ACM TOPLAS* **12**, 123–34.

Glauert, J.R.W. (1992) Asynchronous mobile processes and graph rewriting. *In Proc. PARLE'92*, Champs sur Marne, Paris, June 1992. Lecture Notes in Computer Science **605**, Springer-Verlag, Berlin, pp. 63–78.

Glauert, J.R.W. and Papadopoulos, G.A. (1988) A parallel implementation of GHC. *In Proc. FGCS88*, Tokyo, Japan, 28 November–2 December , Vol. 3, pp. 1051–8.

Glauert, J.R.W., Hammond, K., Kennaway, J.R. and Papadopoulos, G.A. (1988a) Using Dactl to implement declarative languages. *In Proc. Conpar88*, Manchester, UK, 12–16 September, Cambridge University Press, Cambridge, pp. 116–24.

Glauert, J.R.W., Kennaway, J.R., Sleep, M.R. and Somner, G.W. (1988b) Final specification of Dactl. Report SYS-C88-11, School of Information Systems, University of East Anglia, Norwich, UK.

Glauert, J.R.W., Leth, L. and Thomsen, B. (1993) A new process model for functions in term graph rewriting. Chapter 18 *in Term Graph Rewriting: Theory and Practice*, (eds M.R. Sleep, M.J. Plasmeijer and M.C.J.D. van Eekelen), Wiley, New York.

Giacalone, A., Mishra, P. and Prasad S. (1989) Facile: a symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming* **18**, 121–60.

Gregory, S. (1987) *Parallel Logic Programming in PARLOG – The Language and its Implementation*. Addison-Wesley, London.

Hammond, K. (1990) *Parallel SML: A Functional Language and its Implementation in Dactl*. Pitman, London (Ph.D. Thesis, School of Information Systems, University of East Anglia, Norwich, UK).

Hammond, K. and Papadopoulos, G.A. (1988) Parallel implementations of declarative languages based on graph rewriting. *In Proc. IT 88*, Swansea, UK, 4–7 July, pp. 246–9.

Hoffmann, C. and O'Donnell, M.J. (1982) Programming with equations. *ACM Transactions on Programming Languages and Systems* **4**(1), 83–112.

Hoffmann, B. and Plump, D. (1988) Jungle evaluation for efficient term rewriting *In* Proc. Joint Workshop on Algebraic and Logic Programming. *Mathematical Research* **49**, pp. 191–203.

Kennaway, J.R. (1988) The correctness of an implementation of functional Dactl by parallel rewriting. *In Proc. IT 88*, Swansea, UK, 4–7 July, pp. 254–7.

Kennaway, J.R. (1990a) Implementing term rewrite languages in Dactl. *Theoretical Computer Science* **72**, 225–50.

Kennaway, J.R. (1990b) Graph rewriting in a category of partial morphisms. *In Proc. Fourth International Workshop on Graph Grammars, Bremen*, Lecture Notes in Computer Science **532**, Springer-Verlag, Berlin, pp. 490–504.

Kennaway, J.R. and Sleep, M.R. (1983) Syntax and informal semantics of DyNe. *In The Analysis of Concurrent Systems*, Lecture Notes in Computer Science **207**, Springer-Verlag, Berlin, pp. 222–30.

Klop, J.W. (1990) Term rewriting systems. Chapter 6 *in Handbook of Logic in Computer Science*, 1, (eds S. Abramsky, D.Gabbay and T. Maibaum), Oxford University Press, Oxford.

Lafont, Y. (1990) Interaction nets. *In 17th ACM POPL*, San Francisco, CA, 17–19 January, ACM Press, pp. 95–108.

Papadopoulos, G.A. (1989) Parallel implementation of concurrent logic languages using graph rewriting techniques. Ph.D. Thesis, University of East Anglia, Norwich, UK.

Papadopoulos, G.A. (1996) Concurrent object-oriented programming using term graph rewriting techniques. *Information and Software Technology*, **38**(8), 539–47.

Papadopoulos, G.A. (1997a) Implementing concurrent logic and functional languages in Dactl. *Journal of Programming Languages* **5**, 99–123.

Papadopoulos, G.A. (1997b) Object-oriented term graph rewriting. *International Journal of Computer Systems Science and Engineering*. To appear.

Peyton Jones, S.L. (1987) *The Implementation of Functional Programming Languages*. Prentice-Hall, Englewood Cliffs, NJ.

Peyton Jones, S.L. and Salkild, J. (1989) The spineless tagless G-Machine. *In Proc. FPCA89*, Addison-Wesley, Reading, MA, pp. 184–201.

Raoult, J.C. (1984) On graph rewritings. *Theoretical Computer Science* **32**, 124.

Rosenfeld, A. and Milgram, D.L. (1972) Web automata and web grammars. *Machine Intelligence* **7**, 307–24.

Shapiro, E.Y. (1989) The family of concurrent logic programming languages. *Computing Surveys* **21**(3), 412–510.

Sleep, M.R., Plasmeijer, M.J., and van Eekelen, M.C.J.D. (eds) (1993) *Term Graph Rewriting: Theory and Practice*. Wiley, New York.

Staples, J. (1980) Computation on graph-like expressions. *Theoretical Computer Science* **10**, 171–85.

Staples, J. (1980) Optimal evaluations of graph-like expressions. *Theoretical Computer Science* **10**, 297–316.

Wadsworth, C.P. (1971) Semantics and pragmatics of the lambda-calculus. Ph.D. thesis, University of Oxford.

Watson, I., Woods, V., Watson, P., Banach, R., Greenberg, M. and Sargeant, J. (1988) Flagship: a parallel architecture for declarative programming. *In Proc. 15th ISCA*, Hawaii, 30 May–2 June, ACM Press, pp. 124–30.

Appendix: Dactl syntax

A.1 Symbols

The lexical syntax will be given using the following definitions:

Upper An upper-case letter.

Lower A lower-case letter.

Digit Any decimal digit.

The lexical classes *IntSymbol*, *LongSymbol*, *RealSymbol*, *CharSymbol* and *StringSymbol* are defined to be respectively, the integer, long integer, real, character, and string constants of the C programming language. Dactl has in addition the classes *Symbol* and *Nodeid*:

$Symbol$ ::= $Upper\{IdentChar\}$.
$Nodeid$ ::= $Lower\{IdentChar\}$.
$IdentChar$ ::= $Upper \mid Lower \mid Digit \mid . \mid \_ \mid ' \mid$ ”.

The lexical class *DataValue* is the union of *IntSymbol*, *LongSymbol*, *RealSymbol*, *CharSymbol*, *StringSymbol* and *PointerSymbol*.

{. . . } indicates zero or more repetitions; [. . .] indicates zero or one repetitions.

A.2 Comments

Any text between an opening brace ({) and a closing brace (}) or end of line is ignored (unless the opening brace is part of a *CharSymbol* or *StringSymbol*).

A.3 Syntax

The reserved words of Dactl are *MODULE*, *ENDMODULE*, *SYMBOL*, *PATTERN*, *IMPORTS*, *PUBLIC* and *RULE*. Capitalization is significant.

Module ::= *MODULE Symbol*; {*Item*} *ENDMODULE Symbol* ; .
Item ::= *IMPORTS ImportsItem* {*ImportsItem*}
| *SYMBOL AccessClass* [*PUBLICAccessClass*] *SymbolList*

```
                    | PATTERN [PUBLIC] PatDefList
                    | RULE RuleList .
ImportsItem     ::= Symbol [ FROM StringSymbol ] ; .
SymbolList      ::= Symbol ; {Symbol ; }.
PatDefList      ::= PatternDef ; {PatternDef ; }.
PatternDef      ::= Symbol = Pattern .
Pattern         ::= NodeDefinition{,NodeDefinition}.
NodeDefinition  ::= [Nodeid : ] Node .
Node            ::= [ NodeMark ] DataValue
                    | [ NodeMark ] Symbol [ [ MarkedTerm {MarkedTerm} ] ]
                    | ( Term {PatternOp Term } )
                    | SymbolClass.
SymbolClass     ::= DataClass
                    | SpecialClass
                    | AccessClass.
DataClass       ::= INT | LONG | REAL | BOOL | CHAR |
                    | STRING | PTR | VECTORC | VECTORO.
SpecialClass    ::= ANY | NONE.
AccessClass     ::= READABLE | CREATABLE | OVERWRITABLE
                    | REWRITABLE | GENERAL .
NodeMark        ::= ∗ | #{#} .
MarkedTerm      ::= [ ArcMark ] Term .
ArcMark         ::= ^ .
Term            ::= Nodeid | NodeDefinition | Activation .
Activation      ::= [ NodeMark ] Nodeid .
PatternOp       ::= + || &.
RuleList        ::= RuleGroup ; {RuleGroup; } .
RuleGroup       ::= Rule {| Rule} .
Rule            ::= Pattern > [ContractumPart{,ContractumPart}]
                    | Pattern ⟹ Term{,ContractumPart} .
ContractumPart  ::= NodeDefinition | Activation | Redirection .
Redirection     ::= Nodeid := Term .
```