USING DACTL TO IMPLEMENT DECLARATIVE LANGUAGES

J.R.W. Glauert,
Declarative Systems Project, School of Information Systems, University of East Anglia, Norwich NR4 7TJ.

K. Hammond,
Declarative Systems Project, School of Information Systems, University of East Anglia, Norwich NR4 7TJ.

J. R. Kennaway,
Declarative Systems Project, School of Information Systems, University of East Anglia, Norwich NR4 7TJ.

G. A. Papadopoulos,
Declarative Systems Project, School of Information Systems, University of East Anglia, Norwich NR4 7TJ.

THE DACTL LANGUAGE.

Dactl is a language of graph rewriting, intended as an intermediate language and compiler target language for highly parallel machines. It is based on a form of graph rewriting which can be used to implement functional, logic, and imperative languages.

Graphs.

Graph rewriting is a standard technique for implementing term rewrite languages such as Hope, Standard ML, and Miranda (Miranda is a trademark of Research Software Ltd., Caterbury, UK) [1, 10, 13]. Dactl, however, views graphs as the fundamental objects — terms are merely a view of graphs in which sharing is ignored. Although a formal definition of graph rewriting is more complicated than for term rewriting, we consider graphs to be the more fundamental concept.

A Dactl graph is a directed graph. Each node is labelled with a symbol which may correspond to a function, predicate, or constructor according to the requirements of the programmer. From each node originates an ordered set of arcs leading to successor nodes. Graphs may be cyclic and need not be connected. However, 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 of interest.

Graphs are written in a form similar to terms, but using repetition of defined variables to indicate sharing. Here are some examples of Dactl graphs, in textual and diagrammatic form:

```
Append[ s:Cons[ 0 Nil ] s ]                    c: Cons[ 1 c ]
```

$$\begin{tikzcd} & \text{Append} \arrow[d, bend right] \arrow[d, bend left] & \\ & \text{Cons} \arrow[dl] \arrow[dr] & \\ 0 & & \text{Nil} \end{tikzcd}$$

$$\begin{tikzcd} & \text{Cons} \arrow[loop right] \arrow[dl] \\ 0 & \end{tikzcd}$$

Rewrite rules.

Dactl rewrite rules generalise term rewrite rules in several ways:

(i) The left-hand side of a Dactl rule is a graph, not a tree. Repeated free variables test "pointer equality", not the textual equality of classical term rewriting (which is never implemented in functional languages based on term rewriting). A left-hand-side graph is rooted, and every node in it must be accessible from the root.

(ii) Whereas a term rule, implemented by graph rewriting, only rewrites the root of the matched subgraph, a Dactl rule may rewrite several nodes, not necessarily including the root of the match.

(iii) In term rewrite languages such as ML, the *evaluation strategy*, which determines which redex to reduce next, is built-in to the language, and is driven by the pattern-matching. In Dactl, it is represented explicitly in the Dactl program.

Expressing the evaluation strategy.

The nodes in the graph are divided into three classes: active, blocked, and idle. *Rule-matching may only be attempted at active nodes.* That is, the root of a left-hand side may only be matched to an active node. Rule-matching may be attempted concurrently at different active nodes.

The right-hand side of a rule specifies the class of each created node. Thus rule-matching an active node may cause new active nodes to be created. Each locus of control thus appoints its own successors. This contrasts with the von Neumann model of computation, where there is a single locus of control appointing a single successor at each step.

A blocked node is waiting for some other computation to terminate before it proceeds. It bears a count (denoted by a string of # signs), and some of its out-arcs will be marked with ^, signifying that the node is waiting for the corresponding nodes to finish. For a node to "finish" means that rule-matching is attempted at that node, but no rule is found to match. The node is then made idle, and a *notify signal* is sent to each of its parents along a ^-marked arc. This signal decrements the blocked count of each such parent; if such a count becomes zero, that parent is made active.

An example shows how this works in practice.

```
Fac[ 0 ]                  =>     *1;
Fac[ n:(Int-0) ]          =>     #IMul[ n ^#Fac[ ^*ISub[ n 1 ] ] ];
Fac[ n:(Any-Int) ]        =>     #Fac[ ^*n ];
```

The first two rules are the natural rules for `Fac`. The activation of `1` in the first rule causes waiting parents to be notified (there being no rules for the symbol `1`). The control markings in the second rule ensure that the arguments to each functions are evaluated before the function becomes active. `IMul` and `ISub` are built-in arithmetic operators. The third rule handles the case where `Fac` is applied to an unevaluated argument. In term rewrite languages, the failure of the first two rules to match causes evaluation of the argument. In Dactl, this is not so: *Dactl pattern-matching never invokes evaluation.* Instead, we must provide for this case explicitly. Here, the third rule activates the argument, and waits for it to notify.

Note that the left-hand sides of rules bear no markings. Other than the requirement that rule-matching must begin from an active node, *Dactl pattern-matching is independent of control markings.*

Summary of syntax.

A Dactl rule consists of four parts: the *pattern*, *contractum*, *redirections*, and *activations*. The pattern is a rooted graph, without control markings, in which every node must be accessible from the root. It may include *pattern operators*, which allow boolean combinations of patterns, and the predefined pattern `Any`, which matches any node. A pattern such as `Int` may be considered to be an infinite sum (0 + -1 + 1 + -2 + ...).

The contractum describes the nodes which are to be created when the rule is executed. Identifiers on the left-hand side may appear in the contractum.

The pattern and the contractum may be listed as a set of definitions of node identifiers:

```
a:#IMul[ n ^b ], b:#Fac[ ^c ], c:*ISub[ n d ], d:1
```

or in an equivalent "packed" shorthand:

```
#IMul[ n ^#Fac[ ^*ISub[ n 1 ] ] ]
```

The longhand form makes it clear that the mark ^ belongs to an arc, but * and # belong to nodes.

Redirections have the form `x:=y`, where `x` is a left-hand side identifier. When the rule is executed, every edge in the graph which points to the node matched by `x` is redirected so as to point to the (matched or new) node `y` (the *target* of the redirection).

Redirection appears at first glance to be impractical, requiring a search of the whole graph. However, redirection to a newly created node can be implemented by overwriting the "redirected" node with the required contents, rather than creating a new node. Redirecting to an existing node may be implemented by the usual method of indirection nodes [12]. The redirection concept allows a uniform treatment of both types of rewrite, and makes the semantics independent of low-level implementation details.

The activations have the form `*x`, where `x` is a left-hand side identifier. This indicates that on executing this rule, the node matched by `x` is to be given the marking `*`, if it was idle. (If `x` already bore a non-idle marking, the activation is ignored.)

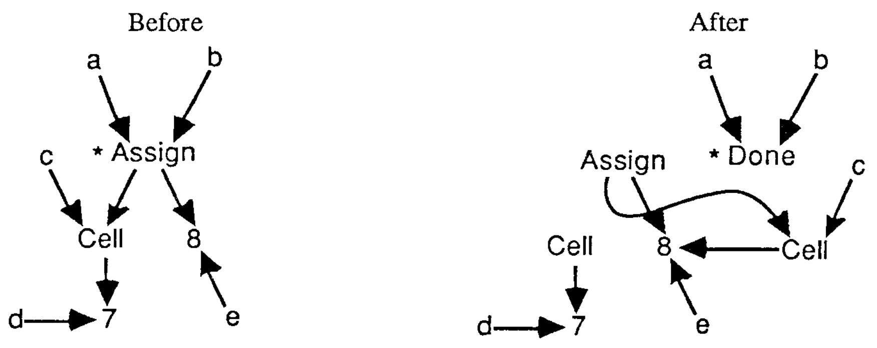
Symbols must be declared, and rules may be grouped into modules, but we will not go into detail on these points. The full definition of Dactl is in [3].

Examples.

Here is an example of a form of rule more general than that found in term rewriting:

```
r:Assign[ z:Cell[ x ] y ]   ->   r:=*Done,  z:=Cell[ y ]
```

Note the single arrow separating the two sides, instead of the double arrow used previously. The double-arrow is actually syntactic sugar, implying a redirection of the root. Here, for clarity, we have written the root-redirection explicitly, together with the extra redirection made by this rule of the `Cell` node. The redirection of the `z` performs the actual assignment: anyone else who had a pointer to that node now has a pointer to the new `Cell` node. Redirection of `r` ensures that the parents of the `Assign` node will be notified that the assignment has been done. Here is a picture of the effect of executing this rule somewhere in a larger graph:



It would not do to redirect `x` to `y`, instead of `z` to a new `Cell`. In the example, this would have the effect that `d` would afterwards point to `8` instead of `7` — rather disconcerting if integers are expected not to spontaneously change their values!

Here is a simple semaphore.

```
r:Wait[ s:FreeSema ]              ->    r:=*Done, s:=EngagedSema;         { W1
Wait[ s:EngagedSema ]             ->    #Wait[ ^s ];                      { W2
r:Signal[ s:EngagedSema  ]        ->    r:=*Done, s:=*FreeSema            { S
```

Rule W1 allows a `Wait` to acquire the semaphore if it is free. Rule W2 causes a `Wait` to continue to wait if the semaphore is engaged. When someone frees and activates the semaphore (by executing rule S) all `Wait` nodes waiting on that semaphore will be notified, and one will succeed in acquiring it.

Serialisability.

What does it mean for multiple rewrites to occur concurrently? Rule-execution at one active node might interfere with rule-execution at another. Dactl requires the following fundamental property to be observed by any implementation: *the result of a Dactl computation must be identical to the result of some sequence of rewrites.* This serialisability constraint is essential if there is to be any hope of proving useful properties of a Dactl program. In general, when agents are performing multiple rewrites, and rule-matching may inspect parts of the graph being rewritten by other agents, some sort of locking protocol must be used to ensure serialisability of rewrites. However, for certain subsets, we are able to prove that only a small amount of locking is actually necessary, even when the underlying hardware only serialises operations affecting single nodes [8].

COMPILING FUNCTIONAL LANGUAGES.

Due to space constraints we can only summarise methods of translating functional languages into Dactl. See [6] or [7] for fuller accounts. We consider pure functional languages, though many of the techniques are also applicable to functional languages with side-effects. [7] considers the translation of a parallel version of Standard ML incorporating assignment, input/output and exception handling. We consider both eager and lazy reduction strategies.

Simple translation.

A functional program consists of a set of function definitions, and a goal expression to be evaluated by using those definitions as rewrite rules. This maps easily into Dactl by translating the goal expression into the initial graph, and the function definitions into Dactl rule-sets. Apart from translating the concrete syntax, all that is required is to add suitable control markings to the right-hand sides of the rules.

For an eager evaluation strategy, the application of control markings can be based on the principle that a graph should rewrite to normal form when it is activated. We can ensure this by activating all the leaves of each right-hand side, marking all arcs on the rhs with ^, and adding an appropriate number of # marks to non-leaf nodes. Thus all arguments to a function will be evaluated in parallel. However, care must be taken with conditional operators such as "if", which must be implemented lazily.

We translate each function name into a Dactl symbol and use polyadic nodes for function application (ignoring partial applications, and those where the function is a general expression). For example, consider the Standard ML definition:

```
fun    ack 0 n = n+1
|      ack m 0 = ack (m-1) 1
|      ack m n = ack (m-1) (ack m (n-1));
```

This translates to

```
Ack[0 n:Int] => *IAdd[n 1];
Ack[m:Int 0] => #Ack[^*ISub[m 1] 1];
Ack[m:Int n:Int] => ##Ack[^*ISub[m 1] ^#Ack[m ^*ISub[n 1]]];
```

`IAdd` and `ISub` are standard arithmetic operators. We have made some optimisations to the simple-minded scheme we described, by not activating things we can see to be in normal form already.

Lazy evaluation.

With a lazy reduction strategy, expressions are only to be reduced to head-normal-form (i.e. having a constructor outermost, but possibly containing unreduced sub-expressions). An output driver (also programmed in Dactl) forces reduction to normal form of the goal expression. Thus translated function definitions must handle partially evaluated forms. The Ackermann function is now translated as:

```
Ack[0 n] => #IAdd[^*n 1];
Ack[m:Int 0] => *Ack[ISub[m 1] 1];
```

```
Ack[m:Int n:Int] => *Ack[ISub[m 1] Ack[m ISub[n 1]]];
Ack[m:Int n:(Any-Int)] => #Ack[m ^*n];
Ack[m:(Any-Int) n:Any] => #Ack[^*m n];
```

The last two rules force the evaluation of unevaluated arguments.

It is easy to prove that if a node is activated, it will not notify unless and until it reaches head-normal form. Each of the first three rules transforms the graph in the same way as one of the original ML rules, hence such a head-normal form must be correct. The last two "default" rules implement a left-to-right pattern-matching strategy, which ensures [12] that head-normal form will be reached whenever possible. The statement and proof of the output driver are equally simple [7].

Since in fact Ackermann's function is strict in both arguments we could use the eager translation in place of the lazy one, to obtain parallelism. Thus strictness analysis, such as in [12], may be used to improve parallelism.

Higher-order functions.

The translations above assume that the root of a function application is a function name. Unfortunately, with higher-order functions this need not be the case. Retaining our translation for the simple case, we introduce a function symbol "AP" to handle more complex applications. Rules for "AP" will handle partial applications:

```
AP[Ack arg1] => *Ack[arg1];
AP[Ack[arg1] arg2] => *Ack[arg1 arg2];
...
AP[f x] => #AP[^*f x];
```

The final rule will not loop infinitely since (assuming that the original functional program was correct) "f" will be reduced to a form which will match one of the other "AP" rules.

Deep pattern-matching.

Deep pattern-matching does not present a problem for an eager reduction strategy. When a function node becomes active, its arguments will all be in normal form. With the lazy translation, firing a node only reduces it to head normal form, which may not be sufficient. We consider two schemes which eliminate this problem.

Default rules.

One solution is to add more default rules, which will force as much evaluation of arguments as is required. For example:

```
fun   sums 0 1 = []
|     sums n (m::(1 as n::_)) = m+n :: sums (n-1) 1
|     sums n 1 = 1;
```

(which forms a list of sums of adjacent pairs of elements — e.g. sums 3 [1,2,3,4,5] = [3,5,7]) translates to

```
Sums[0 1] => *Nil;
Sums[n:Int Cons[m 1:Cons[n Any]]] => *Cons[IAdd[m n] Sums[ISub[n 1] 1]];
Sums[n:Int Cons[m 1:(Any-List)]] => #Sums[n Cons[ m ^*1]];
Sums[n:Int 1:List] => *1;
Sums[n:Int 1:(Any-List)] => #Sums[n ^*1];
Sums[n:(Any-Int) 1:Any] => #Sums[^*n 1];
```

Although this might seem to involve considerable additional computation, in practice a good Dactl implementation should optimise much of the shared matching. The advantage of this scheme is that it is relatively simple to prove conditional correctness.

Pre-evaluation of needed arguments.

We can remove the need for any default rules at all, by transforming the functional program to eliminate deep pattern-matching. The necessary transformations are stated in [9], and proved correct in the extended version of that paper. The example above translates to:

```
Sums[ 0 1 ]  =>  *Nil;
Sums[ n:Int x1 ]  =>  #Sums1[ n ^*x1 ];
Sums1[ n x2:Cons[ first x3 ] ]  =>  #Sums2[ n x2 first ^*x3 ];
Sums1[ n 1:Nil ]  => *1;
Sums2[ n x4 first rest:Cons[ second tail ] ]  =>
         *Cons[ IAdd1[ first second ] Sums3[ ISub[ n 1 ] rest ] ];
Sums2[ n 1 first Nil ]  =>  *1;
Sums3[ x5 x6 ]  =>  #Sums[ ^*x5 x6 ];
IAdd1[ x7 x8 ]  =>  ##IAdd[ ^*x7 ^*x8 ];
```

The pattern-matching required by `sums` is broken down into steps, each of which only requires arguments to be reduced to head normal form.

LOGIC LANGUAGES.

The following PARLOG [5] procedure partitions a list according to the first parameter:

```
mode partition(?,?,^,^).
partition(U,[V|X],[V|X1],X2) <- V<U : partition(U,X,X1,X2).
partition(U,[V|X],X1,[V|X2]) <- U=<V : partition(U,X,X1,X2).
partition(_,[],[],[]).
```

where the output arguments are always variables. This is translated to Dactl as follows:

```
Partition[u Cons[v x] p3 p4] => #Partition_Commit[^g1 ^g2 u v x p3 p4],
                                g1:*Less[v u], g2:*Lesseq[u v]|
Partition[Any Nil p3:Var p4:Var] => *SUCCEED, p3:=*Nil, p4:=*Nil|
Partition[p1 p2:Var p3 p4] => #Partition[p1 ^p2 p3 p4]|
Partition[Any (Any-Var-List) Any Any] => *FAIL;
Partition_Commit[SUCCEED Any u v x p3:Var p4]
                          => *Partition[u x x1 p4],p3:=*Cons[v x1:Var]|
Partition_Commit[Any SUCCEED u v x p3 p4:Var]
                          => *Partition[u x p3 x2],p4:=*Cons[v x2:Var]|
Partition_Commit[FAIL FAIL Any Any Any Any Any] => *FAIL;
r:Partition_Commit[Any Any Any Any Any Any Any] -> #r;
```

The first two clauses of partition are coalesced into a single Dactl rule which performs the common pattern-matching of the second argument only once. The rule rewrites to a function of the general form `Predname_Commit[guards env]`. The function evaluates its guards and commits to the body of the appropriate clause using the environment of the passive part of the clauses. Note that `Predname_Commit` will become active as soon as either of the guards notifies, because it has only a single #; this is a very useful technique for expressing non-determinism in Dactl. If all the guards fail, `Predname_Commit` reports failure and terminates. The last rule is used to suspend the function if some of its guards are still executing.

Overlapping guarded clauses.

Certain guarded clauses possess overlapping input patterns which cannot be coerced to a set of rules with non-overlapping patterns. Such clauses are first transformed to non-overlapping ones by extending them with a new dummy argument having a unique value for each pattern, and then firing all the corresponding rules in parallel using a metarule. The following program fragment (part of a GHC [14] meta-interpreter) illustrates this. The guard in the second clause is assumed to be safe (for implementing full GHC which requires a run-time safety test see [4]).

```
pred(true,X,_)    :-  true | X=ok.
pred(A,X,_)       :-  ghcsystem(A) | X=ok, call(A).
pred((A,B),X,C)   :-  true | pred(A,Xa,Ca), pred(B,Xb,Cb),
                             and(Xa,Ca,Xb,Cb,X,C).
```

This is translated to Dactl as follows:

```
Pred[p1 p2 p3]  => result:Var,   ##OR[^o1 ^o2 result],
                   o1:*Pred'["I1" p1 p2 p3 result],
                   o2:*Pred'["I2" p1 p2 p3 result];
Pred'["I1" True x Any result:Var]  ->  result := *Unify[x "Ok"]|
Pred'["I1" Cl[a b] x c result:Var]  -> result := #AND[^b1 ^b2 ^b3],
                                          b1:*Pred[a xa:Var ca:Var],
                                          b2:*Pred[b xb:Var cb:Var],
                                          b3:*And[xa ca xb cb x c]|
Pred'["I2" a x Any]  =>  #Pred'_Commit[^*Ghcsystem[a] a x result];
Pred'["I1" p1:Var p2 p3 result:Var]  =>  #Pred'["I1" ^p1 p2 p3 result];
Pred'[Any Any Any Any Any]  =>  *FAIL ;
Pred'_Commit[SUCCEED a x result:Var]  -> result := #AND[^b1 ^b2],
                                    b1:*Unify[x "Ok"], b2:*Call[a]|
Pred'_Commit[FAIL Any Any Any]  =>  *FAIL;
```

where

```
AND[SUCCEED SUCCEED ... SUCCEED]  =>  *SUCCEED;
r:AND[(Any-FAIL) (Any-FAIL) ... (Any-FAIL)]  ->  #r;
AND[Any Any ... Any]  =>  *FAIL ;
OR[FAIL FAIL ... FAIL result]  ->  result := *FAIL;
```

Note the use of the auxiliary "variable" `result` to commit to the appropriate body. Nodes such as `result` which can be instantiated to a function (rather than a value), are termed *stateholders* .

Speculative parallelism.

Concurrent logic languages make use of the notion of speculative work: processes will be spawned to do work that in the end may prove to be useless. This can happen when either a guard in an or-conjunction succeeds, or a call in an and-conjunction fails; in both cases further computation in the or- or and-conjunction in question is unnecessary. However, terminating an unnecessary computation by means of low-level implementations of kill signals is not easy in a graph reduction model where it is hard to detect unneeded but active portions of graph. Instead, we simulate the broadcast of a kill signal by means of special variables that may be instantiated to a constructor STOP. These variables are monitored by the Dactl rules which refrain from spawning further computation if they become instantiated. We illustrate this technique with the following PARLOG example:

```
mode ontree(?,^,?).
ontree(key,val,T(x,P(key,val),y)).
ontree(key,val,T(x,P(rkey,rval),y)) <- ontree(key,val,x) : .
ontree(key,val,T(x,P(rkey,rval),y)) <- ontree(key,val,y) : .
```

which is translated to Dactl as follows:

```
Ontree[STOP Any Any Any] => *STOP|
Ontree[k:Var key val Tup["T" x Tup["P" key' val'] y]]
          => #Ontree_Commit[^k c:Var ^g1 ^g2 ^g3 val val' val1 val2],
                       g1:*Eq[key key'],
                       g2:*Ontree[c key val1:Var x],
                       g3:*Ontree[c key val2:Var y]|
Ontree[k:Var key val p:Var] => #Ontree[^k key val ^p]|
Ontree[Any Any Any Any] => *FAIL;
Ontree_Commit[STOP c:Var Any Any Any Any Any Any Any] => *STOP,c:=*STOP|
Ontree_Commit[k:Var c:Var SUCCEED Any Any val val' Any Any]
                                   => *Unify[val val'], c:=*STOP|
Ontree_Commit[k:Var c:Var Any SUCCEED Any val Any val1 Any]
                                   => *Unify[val val1], c:=*STOP|
Ontree_Commit[k:Var c:Var Any Any SUCCEED val Any Any val2]
                                   => *Unify[val val2], c:=*STOP|
Ontree_Commit[Any Any FAIL FAIL FAIL Any Any Any Any] => *FAIL;
r:Ontree_Commit[Any Any Any Any Any Any Any Any Any] -> #r;
```

The variables `k` and `c` propagate the kill signal down the computation tree. The "local" control variable `c` will be instantiated by `Ontree_Commit` if either any of its guards succeeds or it receives a kill signal from some other process via the "global" control variable `k`.

DACTL AS A COMPILER TARGET LANGUAGE.

As a notation for experimenting with reduction strategies in functional languages Dactl is unrivalled. The level of abstraction is sufficient to avoid implementation details which arise when using conventional languages. At the same time the graph rewriting model corresponds closely to the operational model used in most functional language compilers, thus translation is relatively straightforward and focusses attention on optimisations rather than the basic translation scheme. For committed-choice logic languages, most clauses can be transfomed directly into Dactl rules. However, guarded clauses with overlapping input patterns require additional rules for parallel matching since Dactl selects a rule on the basis of matching only. Thus parallel evaluation of guards must be handled explicitly by the translation process. We are currently investigating the extension of our model to accomodate full PROLOG.

Low-level pattern matching facilitates the translation of modern pattern-matching languages and permits experimentation with pattern-matching strategies. Node and arc markings provide fine-grain control over parallel execution where this is available. Garbage collection and low-level resource management is provided by the Dactl compiler and is thus not an issue for the declarative language implementor, although annotations may be used to influence particular implementations.

DACTL FOR PERFORMANCE EVALUATION.

We translated three declarative programs using our translation schemes for logic and functional languages. Our results, using metrics supported by the Dactl reference interpreter, are summarised below. The bold figures refer to the functional version of each program, the remainder refer to the logic version. Parenthesised figures use Dactl redirecion to implement the logic variable directly as opposed to using the unification primitive.

Program	Rewrites	Nodes Created	Parallel Cycles	Avg. Parallelism
Tree Search	291(*278*)	646(*621*)	76(*56*)	5.66(*7.27*)
(31st. of 31 els.)	**276**	**432**	**57**	**7**
Ackermann's	118227(*77121*)	190256(*144056*)	67180(*56868*)	2.75(*2.53*)
function (3,4)	**56509**	**118230**	**61845**	**1.41**
Quicksort	6348(*3692*)	10001(*5796*)	562(*498*)	18.19(*12.15*)
(30 els. reversed)	**2583**	**3889**	**3725**	**1.04**

Generally the results are self-explanatory: the functional implementation shows marginally better performance where the problem is more functional, the logic implementation shows better performance where there is greater scope for speculative parallelism. Note that the logic versions do not implement the kill signal which, in general, would terminate unneeded computation. This is a good contrast between the two declarative models. The dramatic variance in parallelism for the case of Quicksort may be due to an overly sequential definition in the functional version.

REASONING ABOUT DACTL PROGRAMS.

We briefly give a very small example of how one may reason about a Dactl program. Consider again the semaphore example. Let w be the number of times rule W2 has been applied to the semaphore, and s the number of times rule S has been applied to it. Then we must prove the invariant: $w = s$ iff the semaphore is free, and $w = s+1$ iff the semaphore is engaged. This may be proved by considering the effect of each rule: S increments s and makes the semaphore free, W2 increments w and makes the semaphore engaged, and the other rules do not change s, w, or the state of the semaphore. (We must assume that the rest of the program within which these rules occur also has the last property.) At the beginning of the computation, $s = w = 0$ and the semaphore is free, therefore the invariant holds throughout.

We cannot prove fairness, since Dactl itself makes no commitment to fair scheduling of active nodes. The best we can do is to prove that if someone is waiting for the semaphore (i.e. if somewhere in the graph we have `#Wait[^s:EngagedSema]`), and if someone else is ready to release it (i.e. the graph also contains `*Signal[s]`), then it is possible for the semaphore to be released and for one of those waiting to acquire it, and this will remain possible until it happens. A formal definition and proof of this wordy statement would require a foray into temporal logic.

To assist proofs and efficient implementations, Dactl allows annotations by which the user (or, say, a strictness-analysing program) may make assertions about the ways in which the symbols are used.

CONCLUSIONS.

We have introduced Dactl, a compiler target language based on graph-rewriting, and described compilation schemes for logic and functional languages. The relative performance of our schemes have been assessed with the Dactl reference interpreter. At the same time, it appears practical to reason about Dactl code. Thus Dactl is a realistic choice of compiler target language.

REFERENCES.

1. Barendregt, H.P., *et al. Term Graph Rewriting*, Proc. PARLE Conference II, LNCS **259**, 141-158, 1987.

2. Burstall, R.M., MacQueen, D.B., and Sannella, D.T., *HOPE: an Experimental Applicative Language.* Report CSR-62-80, University of Edinburgh, 1980.

3. Glauert, J.R.W., Kennaway, J.R., and Sleep, M.R., *Specification of Dactl1*, School of Information Systems, University of East Anglia, 1987.

4. Glauert, J.R.W. and Papadopoulos, G. A., A *Parallel Implementation of GHC*, FGCS'88, Tokyo, Japan, 1988.

5. Gregory, S., *Parallel Logic Programming in PARLOG: the Language and its Implementation*, Addison-Wesley, London, 1987.

6. Hammond, K., and Papadopoulos, G.A. *Parallel Implementations of Declarative Languages Based on Graph Rewriting*, Alvey Technical Conference, Swansea, 1988.

7. Hammond, K. *Implementing Functional Languages on Parallel Machines*, Ph.D. Thesis, University of East Anglia, in preparation, 1988.

8. Kennaway, J.R. *The Correctness of an Implementation of Functional Dactl by Parallel Rewriting*, Alvey Technical Conference, Swansea, 1988.

9. Kennaway, J.R. *Implementing Term Rewrite Languages in Dactl*, Proc. CAAP'88, LNCS **299**, 102–116, 1988; extended version submitted to Th. Comp. Sci., 1988.

10. Milner, R. *The Standard ML Core Language*, Report CSR-168-84, Edinburgh University, 1984.

11. Papadopoulos, G.A., *A High-Level Parallel Implementation of PARLOG*, Internal Report SYS-88-05, University of East Anglia, 1988.

12. Peyton-Jones, S.L. *The Implementation of Functional Languages*, Prentice-Hall, 1987.

13. Turner, D.A. *Miranda: a Non-strict Language with Polymorphic Types*, FPLCA, LNCS **201**, Springer, 1985.

14. Ueda, K., *Guarded Horn Clauses*, D.Eng. Thesis, University of Tokyo, Japan, 1986.