A FINE GRAIN PARALLEL IMPLEMENTATION OF PARLOG

George A. Papadopoulos

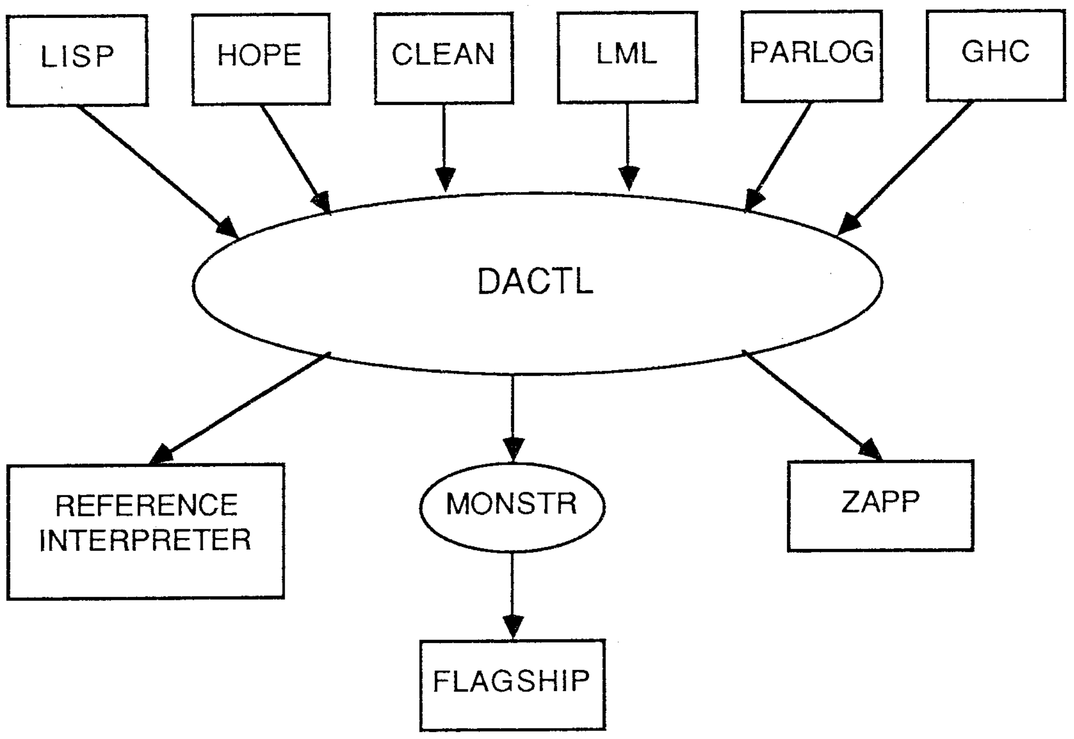
Declarative Systems Project, School of Information Systems,
University of East Anglia, Norwich NR4 7TJ, U.K.
e-mail : gp@sys.uea.ac.uk

Abstract

A parallel implementation of PARLOG is described based on graph rewriting techniques. The implementation is suitable for fine grain parallel architectures such as FLAGSHIP and GRIP. In particular, we compile PARLOG programs to Dactl - an intermediate language based on generalised graph rewriting. We provide a complete translation scheme which maps every PARLOG procedure to a set of Dactl rewrite rules where the only selection process is pattern matching. We show how some of the most subtle features of PARLOG such as non-determinism and metaprogramming can be modelled in the graph rewriting framework of Dactl using suitable rule systems. Finally, we show how our Dactl rules can be translated to MONSTR - a subset of Dactl which is currently implemented on FLAGSHIP. This will allow our PARLOG to Dactl implementation to be directly executable by that machine.

1. Introduction

In this paper we continue our investigation in implementing logic languages in a graph reduction framework[6] by describing a parallel implementation of PARLOG[13] using Dactl (Declarative Alvey Compiler Target Language); Dactl[10] is a computational model based on generalised graph rewriting where programs are written as sets of rewrite rules. The language is intended to serve as a common intermediate language between logic and functional languages and novel computer architectures[20]. Note that Dactl has some similarities with the languages Lean[3] and Clean[4] which are both intermediate languages based on graph rewriting, albeit they are designed to support mainly functional languages. Some of the objectives of such a bridging computational model are: to decouple the development of the languages from that of the architectures so that changes in either level should not necessarily affect the other; to reduce the number of required implementations; to provide a means of assessing the potential of languages for parallel execution and in particular to allow the testing of various execution strategies and models so that the most suitable one for each language can be found; to act as a point of reference in comparing the implementation of a certain language with that of another not belonging necessarily to the same class; finally, to free the programmer from the burden of considering low-level and machine-dependent implementation details. The figure below shows some of the languages that have been implemented in Dactl and architectures on which implementations of Dactl are under way:



In particular, we provide a complete translation scheme that maps every PARLOG procedure to a set of Dactl rewrite rules. Our basic translation scheme is similar to the one for GHC[11]; however, it has been optimised and extended to cope with some advanced metaprogramming features available in PARLOG, such as suspending, resuming and terminating computations, handling exceptions, etc.

Dactl is currently implemented on FLAGSHIP[22] and ZAPP[19]. In particular, FLAGSHIP will initially support a restricted subset of Dactl called MONSTR[1]. In this paper, we also show how our unrestricted Dactl rule systems can be translated to MONSTR without compromising their expresiveness.

The rest of the paper is organised as follows: the next two sections introduce the reader to PARLOG and Dactl; the following decribes our PARLOG to Dactl implementation; finally, the last section provides some conclusions and related work.

2. PARLOG

PARLOG is a concurrent logic language featuring *stream and-parallelism* and *committed-choice or-parallelism*; a PARLOG program is a set of guarded clauses of the form

$R(t_1,\dots,t_n)$ <- Guard : Body

together with a mode declaration for every relation R which is of the form

mode $R(m_1,\dots,m_n)$

Each m corresponds to one t and denotes the mode of that argument; a '?' indicates that the argument is *input* and a '^' indicates that it is *output*. To initiate computation, a conjunction of relation calls is used of the form

:- $B_1,\dots,B_m$, n>0

where all the B_m are to be evaluated in parallel. Execution proceeds by attempting to reduce this set of goals using the program clauses; each goal will *commit* to the first clause whose head unifies with that call and whose guard evaluates successfully. That goal then will be substituted by the conjunction of calls in the body of the selected clause. Every PARLOG clause can be translated to its *Kernel* PARLOG equivalent form. Kernel PARLOG is the "unsugared" version of PARLOG; there are no mode declarations and input matching and output unification are done by explicit calls to appropriate primitives in the guard and body respectively for each clause. We have based our implementation on a variant of Kernel PARLOG referred to from now on as Kernel $\text{PARLOG}_{\text{Dactl}}$ or KP_{Dactl} for short; every clause here is of the form

$$R(t_1,\ldots,t_i,t_{i+1},\ldots,t_n) \;\texttt{<-}\; g_1,\ldots,g_p \;:\; b_1,\ldots,b_q.$$

where without loss of generality we assume that the first t_i arguments have input mode and the rest have output mode. The following modifications have been applied to the above clause compared with its original form:

- $t_1,\ldots,t_i$ are arbitrary variable or non-variable terms, where the only restrictions are that the overall input pattern should be linear (i.e. repeated occurrences of variables are not allowed). If a variable v occurs more than once in $t_1,\ldots,t_i$, new variables $v_1,\ldots,v_k$ are introduced for every such subsequent occurrence, and calls to the test unification primitive '==' of the form $v{=}{=}v_1$, $v{=}{=}v_2$, ..., $v{=}{=}v_k$ are added in the guard.
- $t_{i+1},\ldots,t_n$ are distinct variables and for each output argument a call to the full unification primitive '=' is added in the body. If the original program uses the assignment primitive, no explicit call to such a primitive is required; Dactl supports directly the assignment of variables by means of non-root overwrites (see discussion on the unification primitive in the next section).

Although not examined in detail in this paper, the model can cope with the free mixing of the sequential and parallel versions of the clause conjunction and search operators.

3. Dactl

A program in Dactl is a set of rewrite rules specifying a binary *reduction relation* which defines the possible transformations of *graph objects*. Graph rewriting is often used to implement functional languages which have a close resemblance to *term rewriting systems*. Dactl, however, is fundamentally a language of graph rewriting, and although it has been proven that certain, regular, classes of term rewriting systems can be modelled by a graph rewriting language like Dactl[15], the translation of PARLOG uses capabilities not found in term rewriting.

In addition to the specification of a reduction relation, a practical rewriting system must say something about control of evaluation or *reduction strategy*: the choice procedure for selecting candidate redexes from those available in the graph. Dactl can model very general and potentially ambiguous rewriting systems for which there may be no terminating (normalising) strategy. Since Dactl must accommodate different families of languages with divergent operational semantics (lazy functional languages, "eager" concurrent logic languages) no predefined strategy is adequate; instead, *control markings* are employed to determine the order of reduction.

As an example, the following fragment of Dactl defines some rules for a strict append function:

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

Similar notation is used for rewritable functions, such as `Append`, and data value constructors, such as `Cons`. However, there will be no rules for rewriting `Cons` nodes. Each node has a symbol and a list of arcs to successor nodes.

The first rule says that an `Append` node with `Nil` as first argument is to take the value of the second argument. That argument is activated causing further evaluation if it is a rewritable function. The second rule applies when a `Cons` node is the first argument. The result is a new `Cons` node (bearing the suspension marker, '`#`') whose second argument is a recursive call to `Append`. This call is activated, using the '`*`' marker, and the notification marker, '`^`', on the argument, causes the `Cons` node to be reactivated when the result has been calculated. Hence the original caller of `Append` will be notified of completion only when the argument to `Cons` has been evaluated. Note that by changing the second rule to

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

we get an eager append function, and by changing it to

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

we get a lazy one.

In general, Dactl rules take the form:

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

The *pattern* may be matched against any suitable part of the graph; it can be a simple graph or it can contain *pattern operators*. In particular, there are four pattern operators: '`+`', '`-`', '`&`' and `Any`. The intention is that `p+q` matches anything matched by `p` or `q` (union), `p-q` matches anything matched by `p` but not by `q` (difference), `p&q` matches anything matched by both `p` and `q` (intersection) and `Any` matches successfully against any node.

The *contractum* specifies a new graph structure which may contain references to the pattern graph. After a successful matching, a copy of the contractum is built, adding the new structure to the graph. The *redirections* part indicates how the new structure is to be linked into the original graph. A redirection involves a source node identifier (which must be from the original graph) and a target node identifier (usually in the new graph). All references to the source node are changed to become references to the target node. Hence arcs are redirected away from the source to the target.

The example was given in the *shorthand form* of Dactl. The *longhand form* is as follows:

```
a:Append[n y], n:Nil, y:Any -> *y, a:=y|
a:Append[c y], c:Cons[h t], y:Any, h:Any, t:Any -> d:#Cons[h ^b], b:*Append[t y],
                                                   a:=d;
```

The longhand form gives an explicit tabulation of the graphs representing pattern and contractum. The components of a rule are made visible, including the root redirection implied by the use of the '=>' separator between pattern and contractum of rules.

Contractum nodes may be created active, using the '*' marking, or suspended using a marking of the form '#', '##', ... when rewriting of the node will only be considered when a number of children bearing the notification marking '^' equal to the number of '#' markings have responded. Note here that the number of '#' is allowed to be less than the number of children bearing the notification marking; this can be used to express non-strictness. The *activations* section allows a rule to make active some nodes in the original graph which were matched by the pattern.

Only activated nodes will be considered for matching; if a match is found, the corresponding contractum is built and the redirections and activations are performed. However, if no rule matches, we notify all nodes suspended on the node in question by removing a '#' annotation, making the nodes active when the last '#' is removed. This principle of notification on matching failure is rather unfamiliar but explains why many rules will redirect the root of the matched graph to an activated constructor node. Since there are no rules for the constructor, the attempt to match using the constructor will fail and hence those nodes suspended on the constructor will be notified of the result.

Redirection has much the same effect as overwriting the source with the target, and we will often describe the process as overwriting. Although the most frequent kind of redirection has a similar effect to the classical *root-overwrite* of many graph reduction models, Dactl also allows the effect of overwriting non-root nodes. This is particularly important for the PARLOG translation where it is used to model instantiation of logical variables based on the use of a symbol `Var` which is neither a constructor (since it can appear to be overwritten when instantiated) nor a normal function (since there are no rewrite rules for the symbol). Symbols like `Var` are called *overwritables*, as opposed to the *creatable* constructors and the *rewritable* functions. The implementation of a unification primitive, for example, has among others the following rules:

```
Unify[v1:Var v2:Var] => *SUCCEED, v1:=v2|
Unify[v:Var t:(Any-Var)] => *SUCCEED, v:=*t|
Unify[t:(Any-Var) v:Var] => *SUCCEED, v:=*t;
```

plus rules for decomposing data structures. Note here the use of ':=' in the rhs which models the instantiation of variables.

A rule wishing to suspend evaluation until a variable is instantiated creates a suspended node with a notification marker on an arc to the variable node, but does not activate the variable node itself. When another part of the computation wishes to instantiate the variable, it redirects arcs to the variable to the value to be given and it activates the value. If the value is a constructor, matching will fail, and all nodes suspended on the original variable will be notified. Note that in the definition of `Unify` above, we refrained from firing the term a variable is redirected to if it is another variable; thus any nodes suspending on the former variable will not be activated only to suspend again. In addition, a form of rule ordering is available; rules separated by a '|' may be tried for matching in any order whereas rules following a ';'

will only be considered if none of the earlier rules apply. The sequenced form can be considered a shorthand version of an equivalent set of rules using pattern difference operators instead.

A complete description of Dactl is beyond the scope of this paper and can be found in [8,10]. In addition, [2] discusses its theoretical background and [9,14] examine the potential of Dactl for implementing declarative languages.

4. Translating PARLOG to Dactl

4.1 The Basic Translation Scheme

Consider the following PARLOG program which appends two lists:

```
mode append(?,?,^).
append([H|X],Y,[H|Z]) <- append(X,Y,Z).
append([],Y,Y).
```

When translated to KP_{Dactl} gives the following code:

```
append([H|Z],Y,Z) <- Z=[H|Z'], append(X,Y,Z').
append([],Y,Z) <- Z=Y.
```

The equivalent Dactl program is shown below:

```
Append[Cons[h x] y z] => #AND[^*Unify[z Cons[h z':Var]] ^*Append[x y z']]|
Append[Nil y z] => *Unify[z y]|
Append[x:Var y z] => #Append[^x y z]|
Append[(Any-Cons[Any Any]-Nil) Any Any] => *FAIL;
```

Note the use of the pattern operators available in Dactl to select the appropriate rule for matching; note also the introduction of new variables in the rhs of the rules as nodes with the pattern `Var`. If the first input argument is a list, the first rule is selected which spawns two processes: one for the output unification and one for the recursive call. The two processes are monitored by the AND process which remains suspended until *either* of them reports back (note here the use of a single '#'). If AND receives a SUCCEED message from one of the processes it suspends again waiting for the other to complete; if, however, it receives a FAIL message it immediately reports failure and terminates. This early detection of failure allows us to perform an important optimisation: when AND receives a failure message, it terminates the other processes in its conjunction whose computations are now unnecessary; the exact way the killing of active computation is achieved will be explained later on when we describe the implementation of the metacall. The definition of AND is given below:

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

Note the way patterns operators are used in the second rule to detect whether to suspend again or fail. The use of a sequential rule ordering guarantees that the second rule will be tried only when the first does not match. Hence if the second rule matches, the produced pattern is actually

```
AND[(Any-FAIL) (Any-FAIL) ... (Any-FAIL)]-AND[SUCCEED SUCCEED ... SUCCEED]
```

which means that some but not all of the children processes have terminated successfully and thus AND should suspend again. Note here that the rhs '-> #r' expresses exactly this functionality: the root packet for AND will suspend waiting for the remaining children processes to terminate. The equivalent rule

```
AND[p1:(Any-FAIL) p2:(Any-FAIL) ... pn:(Any-FAIL)] => #AND[^p1 ^p2 ... ^pn];
```

would require the creation of a new root packet for AND which in this case is unnecessary. Note also that the technique of using pattern operators to determine whether to suspend or fail is independent of the number of arguments of AND. In general, the detection of suspensions is one of the trickiest issues in implementing concurrent logic languages in a pattern matching based language like Dactl.

Consider now the following program:

```
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(_,[],[],[]).
```

whose KP_{Dactl} code is shown below:

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

The following issues arise in the translation of the above program to Dactl: the handling of the identical input patterns, and the evaluation of guards before commiting to a clause. The equivalent Dactl program is shown below:

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

Partition_Commit[SUCCEED Any u v x x1 x2] => #AND[^b1 ^b2],
                           b1:*Unify[x1 Cons[v x1':Var]], b2:*Partition[u x x1' x2]|
Partition_Commit[Any SUCCEED u v x x1 x2] => #AND[^b1 ^b2],
                           b1:*Unify[x2 Cons[v x2':Var]], b2:*Partition[u x x1 x2']|
Partition_Commit[FAIL FAIL Any Any Any Any Any] => *FAIL;
r:Partition_Commit[Any Any Any Any Any Any Any] -> #r;
```

Since the first two clauses have identical patterns, they can coalesce into a single Dactl rule which performs the required input matching just once. We are then left with two non-overlapping rules; the first solves the two guards and commits to the body of either the first or the second clause, and the second instantiates the output arguments to the empty list if its second argument is also the empty list. In general, guards are solved by a Dactl function that takes the form

```
Predicate_Commit[guard_conjunctions head_and_guard_variables]
```

`guard_conjunctions` is a set of processes, one process for each guard conjunction. If no identical input patterns can be found, then there is one Dactl rule for each clause with an associated `Predicate_Commit` function which, in this case, will have a single guard conjunction. `head_and_guard_variables` are the set of variables appearing in the head of the clause, as well as any new variables appearing in the guards. Thus the global (head variables) and the local (new guard variables) environments are carried forward and used when `Predicate_Commit` commits to the appropriate body.

Finally, we show the translation of guarded clauses with overlapping input patterns. Consider the following program:

```
mode union(?,?,^).
union([X|S1],S2,S) <- member(X,S2,yes) : union(S1,S2,S).
union([X|S1],S2,[X|NewS]) <- member(X,S2,no)  : union(S1,S2,NewS).
union(S1,[X|S2],S) <- member(X,S1,yes)  : union(S1,S2,S).
union(S1,[X|S2],[X|NewS]) <- member(X,S1,no)  : union(S1,S2,NewS).
union([],S,S).
union(S,[],S).

mode member(?,?,^).
member(_,[],no).
member(X,[X|_],yes);
member(X,[_|Y],Answer) <- member(X,Y,Answer).
```

The following issues arise in the translation of the above program to Dactl: the handling of the overlapping input patterns in `union` and the sequential operator in the second clause of `member`. Translating to Dactl gives the following program:

```
Union[p1 p2 p3] => result:Stateholder, ###OR[^o1 ^o2 ^o3 result],
                   o1:*Union'["I1" p1 p2 p3 result],
                   o2:*Union'["I2" p1 p2 p3 result],
                   o3:*Union'["I3" p1 p2 p3 result];

Union'["I1" Nil s2 s result:Stateholder] -> result:=*Unify[s2 s]|
Union'["I1" s1 Nil s result:Stateholder] -> result:=*Unify[s1 s]|
Union'["I2" Cons[x s1] s2 s result:Stateholder]
                               -> result:=#Union'_Commit1[^g1 ^g2 s1 s2 s result],
                                  g1:*Member[x s2 "Yes"], g2:*Member[x s2 "No"]|
Union'["I3" s1 Cons[x s2] s result:Stateholder]
                               -> result:=#Union'_Commit2[^g1 ^g2 s1 s2 s result],
                                  g1:*Member[x s1 "Yes"], g2:*Member[x s1 "No"]|
(Union'[p1 p2 p3 p4 p5]&(Union'[Any Var Any Any Stateholder]+
 Union'[Any Any Var Any Stateholder])) => #Union'[p1 ^p2 ^p3 p4 p5];
Union'[Any Any Any Any Any] => *FAIL;
```

```
Union'_Commit1[SUCCEED Any s1 s2 s result:Stateholder] -> result:=*Union[s1 s2 s]|
Union'_Commit1[Any SUCCEED s1 s2 s result:Stateholder]
                 -> result:=#AND[^*Unify[s Cons[x newS:Var]] ^*Union[s1 s2 newS]]|
Union'_Commit1[FAIL FAIL Any Any Any Any] => *FAIL;
r:Union'_Commit1[Any Any Any Any Any Any] -> #r;

Union'_Commit2[SUCCEED Any s1 s2 s result:Stateholder] -> result:=*Union[s1 s2 s]|
Union'_Commit2[Any SUCCEED s1 s2 s result:Stateholder]
                 -> result:=#AND[^*Unify[s Cons[x newS:Var]] ^*Union[s1 s2 newS]]|
Union'_Commit2[FAIL FAIL Any Any Any Any] => *FAIL;
r:Union'_Commit2[Any Any Any Any Any Any] -> #r;

Member[Any Nil answer] => *Unify[answer "No"]|
Member[x Cons[x' y] answer] => #Member_SEQ[^*Eq[x x'] x y answer]|
Member[p1 p2:Var p3] => #Member[p1 ^p2 p3]|
Member[Any (Any-Var-Cons[Any Any]-Nil) Any] => *FAIL;

Member_SEQ[SUCCEED Any Any answer] => *Unify[answer "Yes"]|
Member_SEQ[FAIL x y answer] => *Member[x y answer];

OR[FAIL FAIL FAIL result:Stateholder] -> result:=*FAIL;
```

The above translation deserves some explanation. Since all the overlapping clauses of `union` must be tried in parallel (we recall here that in Dactl pattern matching is the only selection process and there is no backtracking), we first transform the overlapping patterns to non-overlapping ones by extending them with a dummy argument which is given a unique value for each pattern. Various optimisations are possible here and some of them are illustrated in the translation above: the two unguarded clauses have been given the same argument; the same applies for clauses 1 and 2, and 3 and 4 which have identical input patterns. We are then left with 3 groups of clauses which are fired in parallel monitored by `OR`; these groups share the special node `result` which will be instantiated by one of the processes to the body of the commiting clause. If all the processes report failure, `OR` is activated and rewrites `result` to `FAIL`. Although `result` is instantiated like a normal variable by means of non-root overwrites, the value it is assigned may not be a constructor but rather an executable function; for nodes like `result` we prefer to use the name *stateholder*[1].

Finally note the way sequential search is expressed in Dactl in the translation of `member`.

4.2 Implementation of PARLOG's metacall

PARLOG has been enhanced with a metacall which is used in systems programming and metaprograming. The original three-argument metacall[13] has currently been extended to a five-argument one[7] which has been used to build a sophisticated programming environment[5]. The metacall takes the form

```
call(Module?,Resources?,Goal?,Status^,Control?)
```

which indicates that `Goal` must be solved using the definitions in `Module` with an upper bound of resources defined by `Resources`. `Control` is used by the *metaprogram* (the program executing `call`) to pass to the *object* program (`Goal`) various control messages such as `suspend`, `continue` and `stop` which will cause the suspension, resumption and termination of `Goal` respectfully. `Status` is used by the object program to pass similar messages to its metaprogram as well as exceptions of the form `exception(Type)` and `exception(Type,Goal,NewGoal)`.

A possible implementation of the metacall now follows; this is rather sketchy and a real implementation would involve more rules. For simplicity, we do not consider here the allocation of resources or the handling of modules; both can be implemented quite easily since Dactl is a modular language itself and can be extended with special modules that interface with the underlying implementation. The metacall is represented in Dactl as follows:

```
Call[goal status:Var control:Var signal:Var]
```

where `signal` is used to provide a two-way communication between `Call` and `goal`. The program `append` shown in the previous section can be called using the metacall as follows:

```
Call[^*Append[signal x y z]  status:Var  control:Var  signal:Var]
```

The following Dactl rules implement `Call`:

```
Call[goal:Var status control signal] => #Call[^goal status ^control signal];
Call[goal:Valid_Goal status cont signal] => #Call'[^*goal status ^cont ^signal];
Call[arg status:Var Any Any] => *FAIL, status:=*Exception[Invalid_Arg];
```

The first rule is used to suspend `Call` if its `goal` argument is not instantiated yet. The second rewrites `Call` to `Call'` which then suspends until `goal` terminates or a control message is sent via `control` or `signal` (we assume here suitable definitions for the pattern `Valid_Goal` which determines whether `goal` is a valid goal). The third rule will only match if `goal` is not instantiated to a valid goal in which case `Call` rewrites to `FAIL` and instantiates its `status` argument to an appropriate exception message. Incidentally, this is the only time that `Call` may fail. `Call'` is defined as follows:

```
Call'[result:(SUCCEED+FAIL) status:Var Any Any] => *SUCCEED, status:=*result|
Call'[Any status:Var c:STOP signal:Var] => *SUCCEED, status:=*c, signal:=*c|
Call'[goal status:Var Cons[c:SUSPEND control] signal:Var]
                                        => #Call'[goal status':Var ^control signal],
                                           status:=*Cons[c status'], signal:=c|
Call'[goal status:Var Cons[c:CONTINUE control] signal:Var]
                                        => #Call'[goal status':Var ^control signal],
                                           status:=*Cons[c status'], signal:=c|
Call'[goal status:Var control:Var signal:Cons[exception signal']]
                                           => #Call'[goal status' ^control signal'],
                                              status:=*Cons[exception status':Var];
```

If `goal` completes execution, the first rule is used to terminate `Call'` and instantiate `status` to the result of the computation. If `control` is instantiated by the metaprogram to `STOP`, `Call'` terminates again after instantiating both `status` and `signal` to `STOP`. Thus `goal` which also shares `signal` receives the signal and terminates. The third and fourth rules are used to pass to `goal` the messages `SUSPEND` and `RESUME` which will then react appropriately. Finally, the last rule is executed if `goal` has itself reported an exception; `Call'` will report the exception to its metaprogram and wait for further instructions.

The way `goal` processes the control signals it receives from `Call'` is illustrated by showing its implementation in Dactl for the case of `Append` (where we have assumed here for simplicity that its output argument is always a variable - hence we can use a simple redirection rather than a call to `unify`):

```
Append[STOP Any Any Any] => STOP|
Append[Cons[SUSPEND signal] x y z] => Append[signal x y z]|
Append[signal:Var x:Var y z] => #Append[^signal' x y z],
                                signal:=*Cons[Exception[Deadlock] signal':Var]|
Append[signal:Var Cons[h x] y z:Var] => *Append[signal x y z'],
                                        z:=*Cons[h z':Var]|
Append[Var Nil y z:Var] => *SUCCEED, z:=*y;
Append[Any Any Any Any] => *FAIL;
```

Note that the third rule is used to pass an exception to `call'`; note also that the first two rules have no activation markings causing the termination and suspension respectively of computation.

4.3 PARLOG on FLAGSHIP

Dactl semantics insists that all rewrites take place atomically, i.e. they take place as a single indivisible action or not at all. This insistence on atomicity makes it difficult to implement directly on an asynchronous distributed architecture like FLAGSHIP without some dependence on locking, particularly when arguments may be copied among different processors. This led to the definition of a simplified subset of Dactl named MONSTR[1] (an acronym for Maximum of One Non-root Stateholder per Rewrite) which imposes (among others) the following restrictions:

- Nodes must be balanced (i.e. the number of suspensions '#' must be equal to the number of outgoing return arcs - those bearing an '^').
- No more than one argument position in the patterns of rules rooted at a given symbol may refer to an overwritable or stateholder (like `Var`); the rest of the argument positions must refer to constructors or be irrelevant to pattern matching (i.e. bear the special symbol `Any`).

These restrictions mean that a rewrite may be exported to the processor containing the overwritable it refers to, avoiding problems of synchronisation (if variables were copied) or livelock/deadlock (if migration of the same variables was required by many processors simulatenously). The rest of the arguments involved in the matching process can be copied freely and since they are guaranteed to be constructors there are no consistency problems.

Consider now the following PARLOG program:

```
mode merge(?,?,^).
merge([U|X],Y,[U|Z]) <- merge(X,Y,Z).
merge(X,[V|Y],[V|Z]) <- merge(X,Y,Z).
merge(X,[],X).
merge([],Y,Y).
```

whose Dactl equivalent code is shown below:

```
Merge[Cons[u x] y z] => #AND[^*Unify[z Cons[u z':Var]] ^*Merge[x y z']]|
Merge[x Cons[v y] z] => #AND[^*Unify[z Cons[v z':Var]] ^*Merge[x y z']]|
Merge[Nil y z] => *Unify[z y]|
Merge[x Nil z] => *Unify[z x];
(Merge[p1 p2 p3]&(Merge[Var Any Any]+Merge[Any Var Any])) => #Merge[^p1 ^p2 p3];
Merge[Any Any Any] => *FAIL;
```

The above program violates both the MONSTR restrictions listed above: the patterns in the lhs of the rules may have an overwritable in more than one position; in addition, the rhs makes use of the function AND which has unbalanced nodes. The following transformation produces valid MONSTR code without compromising the "semantics" of the original Dactl rule system:

```
Merge[p1 p2 p3] => result:Stateholder_OR, ##OR2[^o1 ^o2 result],
                   o1:*Merge1[p1 p2 p3 result], o2:*Merge2[p1 p2 p3 result];

Merge1[Cons[u x] y z result] => *Merge1_Body1[u x y z result]|
Merge1[Nil y z result] => *Merge1_Body2[y z result]|
Merge1[p1:Var p2 p3 p4] => #Merge1[^p1 p2 p3 p4];
Merge1[Any Any Any Any] => *FAIL;

Merge2[x Cons[v y] z result] => *Merge2_Body1[v x y z result]|
Merge2[x Nil z result] => *Merge2_Body2[x z result]|
Merge2[p1 p2:Var p3 p4] => #Merge2[p1 ^p2 p3 p4];
Merge2[Any Any Any Any] => *FAIL;

Merge1_Body1[u x y z result:Stateholder_OR] => *SUCCEED,
                                  result:=Stateholder_AND,
                                  ##AND2[^b1 ^b2 result],
                                  #Is_SUCCEED[^b1 result], #Is_SUCCEED[^b2 result],
                                  b1:*Unify[z Cons[u z':Var]], b2:*Merge[x y z'];

Merge1_Body2[y z result:Stateholder_OR] => *SUCCEED, result:=*Unify[z y];
Merge1_Body2[Any Any Any] => *FAIL;

Merge2_Body1[v x y z result:Stateholder_OR] => *SUCCEED,
                                  result:=Stateholder_AND,
                                  ##AND2[^b1 ^b2 result],
                                  #Is_SUCCEED[^b1 result], #Is_SUCCEED[^b2 result],
                                  b1:*Unify[z Cons[v z':Var]], b2:*Merge[x y z'];
```

```
Merge2_Body2[x z result:Stateholder_OR] => *SUCCEED, result:=*Unify[z x];
Merge2_Body2[Any Any Any] => *FAIL;

OR2[FAIL FAIL result:Stateholder_OR] => *FAIL, result:=*FAIL;
OR2[Any Any result:Stateholder_OR] => *SUCCEED, result:=*SUCCEED;

AND2[SUCCEED SUCCEED result:Stateholder_AND] => *SUCCEED, result:=*SUCCEED;
AND2[Any Any Any] => *FAIL;

Is_SUCCEED[FAIL result:Stateholder_AND] => *FAIL, result:=*FAIL;
Is_SUCCEED[Any Any] => *SUCCEED;
```

Note that the translation of this example to MONSTR is particularly difficult due to the non-deterministic input matching involved; we are not able to examine each input argument in turn sequentially, because the decision on whether to suspend or fail must be taken by examining the pattern as a whole. Note also the rewriting of the stateholder `result` to the value `Stateholder_AND` which allows only one ..._`Bodyi` process to commit (any other similar process would fail to match its last argument which would *not* be now `Stateholder_OR`). Finally, note how the non-determinism which in the original Dactl rules was expressed by means of unbalanced nodes, here is retained by having one process per child of an unbalanced node, with all of them sharing a stateholder (`result`); this stateholder will eventually be instantiated to the result of the computation (as we have just said the same stateholder is used as a mutual exclusion variable in the commitment phase).

By producing these sort of transformations we can translate every Dactl rule system to an equivalent MONSTR one while retaining the expressiveness of the original system. The transformed MONSTR rule system then, will be directly executable by the FLAGSHIP machine. We are currently examining the effectiveness of our Dactl to MONSTR transformation using a simulator for a 4-processor FLAGSHIP machine; results will be reported in a future paper.

5. Conclusions and Related Work

In this paper we presented a parallel implementation of PARLOG in the context of the graph reduction framework. In particular, we showed how PARLOG clauses can be transformed to a set of rewrite rules as expressed by the compiler target language Dactl. We provided a high-level software implementation of PARLOG's metacalls and we illustrated how Dactl's flexibility in manipulating graphs allows for the initiation, suspension, resumption and termination of computations as well as the handling of exceptions. Finally, we described a Dactl to MONSTR transformation which retains the expressiveness of the original rule systems, but it also allows their direct implementation on the FLAGSHIP machine. Two major problems were identified during this process. The first was how to translate procedures where committing to a clause depends on the evaluation of guards rather than input pattern matching. We solved it by representing the rhs of each clause as a Dactl function which after executing its respective guard reports either failure or the corresponding body. The second problem was how to model the notion of suspension during input unification. Since Dactl does not handle in any special way PARLOG's variables (which are represented as ordinary nodes having the symbol `Var`) we associated suspension with the presence of certain properties in the patterns produced, which can be detected using the pattern operators that are available in Dactl (note that the handling of deep patterns is particularly difficult - see, for example, [18]).

Although in this paper we concentrated only on how to translate PARLOG to Dactl, we have also examined similar implementations of other concurrent logic languages. In particular, in [11] we show how to implement GHC[21] using a novel efficient technique for performing the required run-time safety test. A GHC to Dactl compiler written in PARLOG is already operational on the top of the SPM (Sequential PARLOG Machine) system. The compiler will be modified to compile PARLOG programs; this will eventually lead to its bootstrapping in Dactl itself. We are currently investigating possible implementations of Flat Concurrent Prolog[16] and a class of equational (logic+functional) languages[12].

Finally, it is worth pointing out that an implementation of PARLOG on ALICE is reported in [17]; note, however, that that implementation supports only flat PARLOG and makes heavy use of the metacalls. Although PARLOG's metacalls can be supported in Dactl as shown in section 4.2, we do not rely on them in implementing full PARLOG.

Acknowledgements

This work was done under the supervision of John Glauert to whom I am indebted for his continuous advise and encouragement. I gratefully acknowledge the help of Richard Banach at Manchester University in understanding MONSTR and Nic Holt at ICL in indicating possible transformation techniques. The anonymous referees are responsible for many of the corrections and improvements in the final version of the paper but for none of any remaining errors and/or omissions.

References

[1] **Banach R.** and **Watson P.**, *Dealing with State on FLAGSHIP: the MONSTR Computational Model*, CONPAR'88, Manchester UK, Sept. 12-16, 1988.

[2] **Barendregt H. P.**, **Eekelen M. C. J. D.**, **Glauert J. R. W.**, **Kennaway J. R.**, **Plasmeijer M. J.** and **Sleep M. R.**, *Term Graph Rewriting*, PARLE, Eindhoven, The Netherlands, June 15-19, 1987, LNCS 259, Springer Verlag, pp. 141-158.

[3] **Barendregt H. P.**, **Eekelen M. C. J. D.**, **Glauert J. R. W.**, **Kennaway J. R.**, **Plasmeijer M. J.** and **Sleep M. R.**, *Towards an Intermediate Language Based on Graph Rewriting*, PARLE, Eindhoven, The Netherlands, June 15-19, 1987, LNCS 259, Springer Verlag, pp. 159-175.

[4] **Brus T. H.**, **van Eekelen M. C. J. D.**, **van Leer M. O.** and **Plasmeijer M. J.**, *Clean: A Language for Functional Graph Rewriting*, FPLCA'87, Oregon, USA, Sept. 14-17, 1987, LNCS 274, Springer Verlag, pp. 364-384.

[5] **Clark K. L.** and **Foster I. T.**, *A Declarative Environment for Concurrent Logic Programming*, TAPSOFT'87, Pisa Italy, Mar. 1987, LNCS 250, Springer Verlag, pp. 212-242.

[6] **Fasel J. H.** and **Keller R. M.** (eds), *Graph Reduction Workshop,* Santa Fe, New Mexico, USA, Sept. 29 - Oct. 1, 1986, LNCS 279, Springer Verlag.

[7] **Foster I. T.**, *Logic Operating Systems: Design Issues,* 4th International Conference on Logic Programming, Melbourne, Australia, May 25-29, 1987, pp. 910-926.

[8] **Glauert J. R. W.**, *An Introduction to Graph Rewriting in Dactl,* Alvey Technical Conference, Swansea UK, July 4-7, 1988, pp. 250-253.

[9] **Glauert J. R. W.**, **Hammond K.**, **Kennaway J. R.** and **Papadopoulos G. A.**, *Using DACTL to Implement Declarative Languages,* CONPAR'88, Manchester UK, Sept. 12-16, 1988.

[10] **Glauert J. R. W.**, **Kennaway J. R.** and **Sleep M. R.**, *Dactl: a Computational Model and Compiler Target Language Based on Graph Reduction,* ICL Tecnical Journal, May, 1987, 5(3), pp. 509-537.

[11] **Glauert J. R. W.** and **Papadopoulos G. A.**, *A Parallel Implementation of GHC,* FGCS'88, Tokyo, Japan, Nov. 28 - Dec. 2, 1988, Vol. 3, pp. 1051-1058.

[12] **DeGroot D.** and **Lindstrom G.**, *Logic Programming: Functions, Relations and Equations,* Prentice Hall, ISBN 0-13-539958-0, 1986.

[13] **Gregory S.**, *Parallel Logic Programming in PARLOG: the Language and its Implementation,* Addison-Wesley, ISBN 0-201-19241-1, 1987.

[14] **Hammond K.** and **Papadopoulos G. A.**, *Parallel Implementations of Declarative Languages Based on Graph Reduction,* Alvey Technical Conference, Swansea, UK, July 4-7, 1988, pp. 246-249.

[15] **Kennaway J. R.**, *Implementing Term Rewrite Languages in Dactl,* CAAP'88, Nancy, France, Mar. 21-24, 1988, LNCS 299, Springer Verlag, pp. 102-116.

[16] **Kliger S.**, **Yardeni E.**, **Kahn K.** and **Shapiro E.**, *The Language FCP(:,?),* FGCS'88, Tokyo, Japan, Nov. 28 - Dec. 2, 1988, Vol. 2, pp. 763-773.

[17] **Lam M.** and **Gregory S.**, *PARLOG and ALICE: A Marriage of Convenience,* 4th International Conference on Logic Programming, Melbourne, Australia, May 25-29, 1987, pp. 294-310.

[18] **Papadopoulos G. A.**, *A High-Level Parallel Implementation of PARLOG,* Internal Report SYS-C88-05, University of East Anglia, UK, 1988, (to be revised).

[19] **Sleep M. R.** and **Kennaway J. R.**, *The Zero Assignment Parallel Processor (ZAPP) Project,* in Distributed Computing Systems Programme, Peter Peregrinous (ed), London 1984, pp. 250-269

[20] **Thakkar S. S.** (ed), *Selected Reprints on Dataflow and Reduction Architectures,* IEEE, Computer Society Press, ISBN 0-8186-0759-9, 1987.

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

[22] **Watson I.**, **Woods V.**, **Watson P.**, **Banach R.**, **Greenberg M.** and **Sargeant J.**, *Flagship: A Parallel Architecture for Declarative Programming,* 15th International Symposium on Computer Architecture, IEEE, Honolulu, Hawaii, May 30 - June 2, 1988, pp. 124-130.