Towards a Common Implementation for Parallel Logic Languages Based on and Intermediate Compiler Target Language

George A. Papadopoulos and John R. W. Glauert
Declarative Systems Project, School of Information Systems
University of East Anglia, Norwich, UK

ABSTRACT

The work described in this document examines the potential of Dactl as a common implementation language for parallel logic languages. Dactl is a computational model based on graph reduction where programs are written as sets of rewrite rules. The language is intended to serve as a common intermediate compiler target language between logic and functional languages and novel computer architectures. In particular, we show that PARLOG and the flat subset of Guarded Horn Clauses can be easily mapped onto Dactl. We also show that all the features of these languages (back unification, difference lists, stream parallelism, or-parallelism *a la* PARLOG, etc.) are directly supported by, or can be easily implemented in Dactl. We provide a substantial number of examples to support our arguments including the implementation of the non-trivial 'set' and 'subset' primitives of PARLOG.

KEYWORDS

Implementation of Parallel Logic Languages; Graph Reduction Models.

ACKNOWLEDGMENTS

We would like to thank the members of the Declarative Systems Project for providing a stimulating environment and in particular Geoff Somner for providing a reference interpreter for Dactl in no time. The first author is grateful to Kevin Hammond for letting him into the secrets of programming in Dactl.

1. INTRODUCTION

The last two decades we have watched the continuous development and widespread use of *declarative languages*. On the one hand we have *functional* and *equational* languages based on *λ—calculus* and *reduction* and on the other hand we have *logic* languages based on *predicate logic* and *resolution*. Both formalisms have proved their potential for parallel execution. In functional or equational languages (ML, HOPE, KRC, SASL) independent subexpressions can be reduced in parallel and in logic languages body calls can be executed in parallel (*and-parallelism*) and search for candidate clauses can be done in parallel (*or-parallelism*). In the former formalism, function compositions and applications as well as the existence of a unique solution restrict the degree of parallelism to manageable levels. In the latter formalism however, the parallelism involved can cause an exponential growth in the number of active processes and many techniques have been devised to restrict it ([**Papa87a**]). In particular, the so called *parallel logic languages* (PARLOG, Guarded Horn Clauses, Concurrent Prolog) that support *stream* and-parallelism and *commited-choice* or-parallelism stand out as suitable candidates for efficient parallel implementations. These languages have themselves undergone major changes as their use gave insight to better ways of implementing them and new features to be incorporated in them.

At the same time a number of suitable parallel architectures have evolved which aim at supporting the implementation of these declarative languages. Among others we have the Manchester Dataflow Machine, ALICE, ZAPP, GRIP, Flagship, Mago's cellular machine, etc. Again, these proposals have undergone changes as in the case of languages.

Although this continuous change is a proof of development and improvement it also creates some problems, mainly the inability to stabilize the particular implementations. Any major change at the language or architecture level renders the corresponding implementation out of date at least. On the other hand such a high degree of coupling between the language and architecture level forces the language designer to consider the low-level implementation details for the particular architectures concerned. This led to the development of Dactl (Declarative Alvey Compiler Target Language, [**GKSH87**]), a computational model which can act as a bridge between the designers of languages and architectures. The use of Dactl as an intermediate compiler target language not only decouples the development of the language from that of the architecture but it also reduces the number of required implementations (see figures 1 and 2 below).

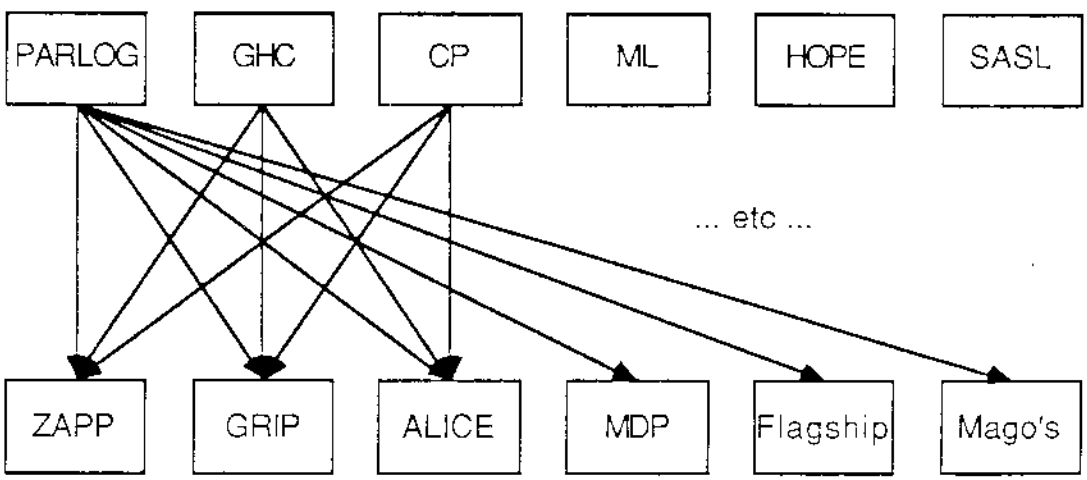

Figure 1

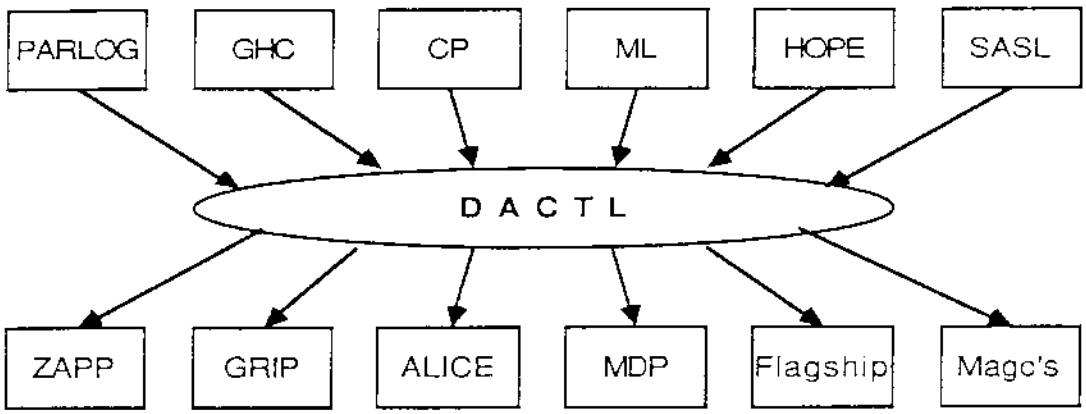

Figure 2

In this report we examine the potential of Dactl for implementing parallel logic languages. In particular, we consider PARLOG and the flat subset of Guarded Horn Clauses (GHC). In section 2 we briefly describe Dactl, PARLOG and GHC, in section 3 we describe the translations of PARLOG and flat GHC to Dactl and in section 4 we comment on our techniques and suggest possible improvements. Then we briefly decribe related work and suggest directions for further research (section 5); finally, we present out conclusions (section 6). The appendices list two non-trivial programs; the first one is the PARLOG 'set' constructor and the second one illustrates the way exhaustive search is done in GHC.

2. THE LANGUAGES

In this section we briefly describe the three languages involved. However the description is essentially condensed and we urge the interested reader to consult the appropriate documents ([**GHSH87**], [**Greg87**], [**Ueda86**]) for more information and details.

2.1 Dactl

Dactl is a graph-rewriting based compiler target language. A Dactl program is a set of rewrite rules which specify a binary *reduction relation* that defines the way a collection of *objects* (the data structures) are manipulated. In Dactl there is no predefined reduction strategy and the language allows the formation of a variety of rewriting systems which we will call *abstract rewriting systems*. Here we are particularly interested in the so called *term rewriting systems* ([**HuOp80**]). It can be shown that any functional program can be mapped onto an equivalent *canonical* term rewriting system, that is one that is *confluent* (*Church-Rosser*) and possibly *noetherian* (*terminating*). On the other hand, a logic program can also be viewed as a set of equivalence-preserving rewrite rules ([**Ders85**]). This is especially true for the parallel logic languages where only one solution is sought and the program can be viewed as such a set of possibly conditional (i.e. guarded) and not necessarily canonical rewrite rules.

In particular, a Dactl program comprises a set of possibly non-linear rewrite rules (where a repeated variable in the left hand side denotes pointer equality) and an *initial graph*. These rules define the *graph transformations* that if applied to the initial graph, a stage will be reached where no more transformations are possible; in this case a normal form has been computed. Each rule is of the form

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

The *pattern* may be matched against any suitable part of the graph; it can be simple or it can contain *pattern operators*. After such a successful matching, a copy of the *contractum*, the right hand side of the rule, is built which specifies new nodes to be added to the graph. The contractum may contain references to the original graph. Secondly, the *redirections* part is performed to redirect some arcs from the original graph to link to the new structure. Although the most frequent kind of redirection is similar to the classical *root-overwrite*, Dactl allows the effect of overwriting any node. As well as dictating the way in which the graph is transformed, a Dactl rule can also control evaluation. Only activated nodes will be considered for matching. New nodes in the contractum may be created active, and also, the final *activations* part of the rule may indicate that some nodes from the original graph are to be made active.

2.2 PARLOG

A PARLOG program is a set of 'guarded clauses' of the form

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

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

mode $R(m_1,\ldots,m_k)$.

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*. During evaluation of a relation call all clauses that define the relation in question are tried in parallel; that is all head unifications are attempted in parallel and all guard evaluations are done in parallel. If during head unification an attempt is made to instantiate a variable in the environment of the call, the call suspends. Upon commitment, full output unification is performed for the 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 unification primitives in the guard and body respectively for each clause.

We have based our implementation on a variant of the *AND tree model* which is close to term rewriting and is fully described in [**Greg87**]. In this model every PARLOG clause is transformed to a set of flat clauses of the form

$R(p_1,\ldots,p_n)$ <- $[g_1\&\ldots\&g_i]$: $[(bp_1\&\ldots\&bp_j)\&](bc_1,\ldots,bc_k)$

where p_1 to p_n are distinct variables, g_1 to g_i are guard primitives, bp_1 to bp_j are body primitives and bc_1 to bc_k are user defined body calls. The brackets are used to denote optional parts. For example the PARLOG `timeslist` program

```
mode timeslist(?,?,^).

timeslist(u,[v|y],[w|z]) <- times(u,v,w), timeslist(u,y,z).
```

translates to the following AND tree program:

```
timeslist(p1,p2,p3) <- DATA(p2) & SET-LIST(p2,v,y) : p3=[w|z]
                       & (times(p1,v,w), timeslist(p1,y,z)).
```

2.3 GHC

GHC is very close to PARLOG; however, whereas in PARLOG synchronization is achieved by means of mode declarations that designate certain arguments of relation calls as either producers or consumers, in GHC the required synchronization is achieved by means of two *rules of suspension* ([Ueda86]):

- Unification invoked directly or indirectly in the head or guard of a clause C called by a goal G cannot instantiate variables in G.
- Unification invoked directly or indirectly in the body of a clause cannot instantiate variables in the guard of C or G until this clause is selected for commitment.

Note that if arbitrary user defined calls are allowed in the guard, there is a need for a (sometimes non-trivial) run-time test to check if the variables involved in unifications are global or local. However if we restrict the language to its flat subset, the resulting sublanguage is very similar to Kernel PARLOG.

3. THE IMPLEMENTATIONS

3.1 PARLOG —> Dactl

We start by showing that all the Kernel PARLOG primitives are directly supported by Dactl. These are:

```
VAR(v)
DATA(v)
BIND(v,t)
GET-CONSTANT(k,v)
GET-LIST(v,v1,v2)
GET-STRUCTURE(F/n,v,v1,...,vn)
```

plus some arithmetic and input/output primitives. The `GET-...` primitives are all pattern matching primitives and since matching is directly supported by Dactl, we simply move the required patterns to the left hand side of appropriate rules. The same holds for the `VAR` primitive since a PARLOG variable is represented in Dactl as a node (`v` say) with the pattern `Var`. The `BIND(v,t)` instructions are represented as redirections of the form `v:=t`. Finally the `DATA(v)` instruction is equivalent to annotating the node that references a variable `v` with the '#' control marking that causes the blocking of the nodes to which it is attached. The node that references `v` then will remain suspended until it becomes instantiated. For example the AND tree code for the `Timeslist` program could be represented as follows:

```
Timeslist[p1 p2:Var p3] => #Timeslist[p1 ^p2 p3];
Timeslist[u Cons[v y] p3] => <... commit ...>;
```

plus rules that define failure to match. However we will be using a different technique as described below.

A PARLOG program is represented as follows: For each procedure there is a top rewrite rule that defines an or-process for that procedure. This process performs the required search for candidate clauses either sequentially or in pseudo-parallel (depending on the search strategy indicated in the original program). It does this by firing a set of rewrite rules, one for each clause of the procedure. Each of these rules has a different outermost function symbol as opposed to the identical relation name of the corresponding PARLOG procedure. This is necessary because two or more clauses may have identical patterns (in which case the guard is used to determine the candidate

clauses) and also to be able to simulate the or-parallelism involved. For example the `fair merge` PARLOG program

```
mode fair-merge(?,?,^).

fair-merge([u|x],y,     [u|z]   ) <- var(y) : fair-merge(x,y,z).
fair-merge(x,     [v|y],[v|z]   ) <- var(x) : fair-merge(x,y,z).
fair-merge([u|x],[v|y],[u,v|z]) <-           fair-merge(x,y,z).
fair-merge([],     y,     y     ).
fair-merge(x,      [],    x     ).
```

has the following top rule:

```
FMerge[p1 p2 p3] => *PAR_search[Cons[FMerge1 Cons[FMerge2
                                Cons[FMerge3 Cons[FMerge4
                                Cons[FMerge5 Nil]]]]] P[p1 p2 p3];
```

For each PARLOG clause there are three rewrite rules: the first one models the successful commitment to that clause, the second models suspension on uninstantiated input arguments and the third reports failure to match. In particular, the left hand sides of these rules comprise the following three patterns respectively: the first one is identical to the one in the corresponding clause (*success pattern*). The second determines whether any input arguments are still uninstantiated (*suspension pattern*). This is achieved by subtracting the success pattern from a more general one comprising for each input argument not only its input pattern but also the pattern `Var`. If the suspension pattern matches the graph then some input variables have not been instantiated yet. Finally the third one determines failure to match (*failure pattern*). Note the use of the pattern operations available in Dactl. For example the third rule of the above program is represented as follows:

```
PATTERN FM31=FMerge3[Cons[Any Any] Cons[Any Any] Any];
        FM32=(FMerge3[(Var+Cons[Any Any]) (Var+Cons[Any Any]) Any] - FM31);

RULE
FMerge3[Cons[u x] Cons[v y] p3]
                          => *Commit[SUCCEED body],
                             body:Conj[bprim bcall],
                             bprim:Unify[p3 Cons[u Cons[v z:Var]]],
                             bcall:FMerge[x y z];
(FMerge3[p1:(Var+Cons[Any Any]) p2:(Var+Cons[Any Any]) p3]-FM31)
                          => *Suspv[Cons[p1 Cons[p2 Nil]]];
(FMerge3[Any Any Any]-FM31-FM32) => *FAIL;
```

where `Suspv` is a function that determines which arguments are still uninstantiated. It does this by examining the list of input arguments. If it comes across a non-variable term it simply discards it; the ones that remain in the end form the list of *suspension variables* as described in **[Greg87]**. In the next section we will see how a clause can directly suspend on unistantiated variables. Note the representation of new variables in the right hand side as new nodes with the pattern `Var`. Note also that when there exists only one input pattern the following obvious optimisations are possible as illustrated using the fourth rule of `fair merge`:

```
FMerge4[Nil p2 p3] => *Commit[SUCCEED body], body:Unify[p3 p2];
FMerge4[v:Var Any Any] => *Cons[p1 Nil];
FMerge4[(Any-Var-Nil) Any Any] => *FAIL;
```

If the input pattern is more than one level deep, there may be a need for additional rules. For example, the following clause

```
mode clause(?).

clause([P(x,y) rest]) <- …
```

has (among others) the following rewrite rules:

```
Clause[p:Var] => *Cons[p Nil];
Clause[Cons[p:Var rest]] => *Cons[p Nil];
Clause[(Any-Var-Cons[Var Any]-Cons[P[Any Any] Any])] => *FAIL;
```

However, in practice deep patterns are rarely used in parallel logic languages (as opposed to what is happening in their functional counterparts).

The right hand sides of the three rewrite rules are defined as follows: the first one comprises the guard and body of the respective PARLOG clause, the second computes the list of suspension variables as explained above and the third simply returns the value `FAIL`. In particular, the right hand side of the first rule is of the form

```
RHS =>   *Commit[guard body],
         guard:  list_of_guard_instructions |
                 SUCCEED (if the guard is empty)
         body:   body_calls (as explained below) |
                 Conj[list_of_primitives body_calls] (if there are body primitives)
```

If `Commit`'s first argument is `SUCCEED` (that is there are no guard calls), it simply returns the body of the clause. Otherwise `Commit` tries to solve `guard`; if it succeeds it returns `body`. Otherwise at least one call in `guard` has either failed in which case it returns `FAIL` or suspended in which case it returns a list of suspension variables. In the latter case no other call in the conjunction must have failed. `body_calls` can be either a single call in which case it is represented explicitly as shown above for the `fair merge` program or a parallel conjunction of calls in which case it is represented as `Body[b1 b2 ... bn]`, one `b` for each body call. We therefore have a set of rewrite rules of the form

```
Body[b1 b2 ... bn] => #And_process[^@b1 ^@b2 ... ^@bn];
```

one for each value of n. Since n can be arbitrary large, we assume the existence of a small number of particular rules (say 5-7); for n greater than 5 or 7 we build a tupled representation using a tree of `Body` nodes. Dactl programs have a module structure and an efficient implementation could provide a special module to handle terms of arbitrary arity. This module could act as an interface to special-purpose hardware to assist commonly used rules.

Each `And_process` rule is of the form

```
And_process[SUCCEED SUCCEED ... SUCCEED] => *SUCCEED;
(And_process[p1:(Any-FAIL) p2:(Any-FAIL) ... pn:(Any-FAIL)]-A1)
                                          => #And_process[^p1 ^p2 ... ^pn];
(And_process[Any Any ... Any]-A1-A2) => *FAIL;
```

where

```
A1=And_process[SUCCEED SUCCEED ... SUCCEED];
A2=(And_process[(Any-FAIL) (Any-FAIL) ... (Any-FAIL)]-A1);
```

Note that each `And_process` rule has a single blocking mark but an arbitrary number of notification marks. Thus, every time an `And_process` receives a signal it activates itself to examine it; if it is a `SUCCEED` it suspends again except in the case when all children have reported success in which case it reports success to its parent and terminates. If a `FAIL` message arrives from one of the children it reports failure and terminates. This early detection of failure is the equivalent of lazy evaluation in functional languages.

The same technique has been used in the implementation of the control metacall. A call of the form `call(`p_1`,`p_2`,...,`p_n`,s,c)` is represented as follows:

```
Call[p1 p2 ... pn s c] => #EVAL[^@p1 ^@p2 ... ^@pn s ^c];
EVAL[Any Any ... Any s:Var STOP] => *SUCCEED, s:=*STOP;
EVAL[SUCCEED SUCCEED ... SUCCEED s:Var (Any-STOP)] => *SUCCEED, s:=*SUCCEED;
(EVAL[p1:(Any-FAIL) p2:(Any-FAIL) ... pn:(Any-FAIL) s c:(Any-STOP)]-Ev1)
                                          => #EVAL[^p1 ^p2 ... ^pn s ^c];
(EVAL[Any Any ... Any s:Var (Any-STOP)]-Ev1-Ev2) => *SUCCEED, s:=*FAIL;
```

where

```
Ev1=EVAL[SUCCEED SUCCEED ... SUCCEED Any (Any-STOP)];
Ev2=(EVAL[(Any-FAIL) (Any-FAIL) ... (Any-FAIL) Any (Any-STOP)] - Ev1);
```

`EVAL` is behaving in a similar way to `AND_process`; however it is also suspended on the control variable `c`. If this is instantiated to `STOP` it terminates immediately. Note the use of non-root overwrite for instantiating the variable `s` to the appropriate value.

These rewrite rules are applied as follows: as explained above for each PARLOG clause there is a top rule of the form

```
Pred[p1 p2 ... pn] => *SEQ_search[list_of_clauses P[p1 p2 ... pn]];
```

or

```
Pred[p1 p2 ... pn] => *PAR_search[list_of_clauses P[p1 p2 ... pn]];
```

where `SEQ_search` and `PAR_search` simulate sequential and parallel search respectively. A `SEQ_search` process eventually reaches a form similar to

```
#SEQ_search[^fired_clause rest_of_clauses]
```

If `fired_clause` returns `FAIL`, `SEQ_search` tries the next one; if it returns the body of the clause `SEQ_search` commits to that and disregards all the other clauses. Finally, if a list of suspension variables is returned, `SEQ_search` suspends on them until one of them is instantiated in which case it retries `fired_clause`. There is a special function (`Suspend`) that monitors the uninstantiated variables. On the other hand a `PAR_search` process reaches a form similar to

```
#PAR_search[^fired_clause rest_of_clauses susp_vars_list]
```

It is operating in a similar way to `SEQ_search`; however, when a list of suspension variables is returned it appends it to `susp_vars_list` and tries the next clause. Only when all the clauses have been tried and no body has been returned, it suspends on the global list of uninstantiated variables. We recall that the list of uninstantiated variables comes from two sources: the first is the `Suspv` call in the second rule of each PARLOG clause; the second is the `Commit[guard body]` call in the first rule.

Appendix I lists the Dactl code for the 'set' constructor primitive whose PARLOG code can be found in [**ClGr85**]. The 'set' primitive (which is in effect an or-interpreter) computes eagerly all the solutions to a query. A call of the form

```
set(solution_list^,term?,conjunction?)
```

will incrementally bind `solution_list` to a list of all the instances of `term` that correspond to different solutions to `conjunction`. In particular, an or-process will be spawned for every clause that matches the first goal of `conjunction`; any solutions for that goal are made immediately available to the rest of the goals in the same conjunction. Thus the 'set' primitive supports or-parallelism and an induced form of and-parallelism. Rules as well as facts are defined as in standard Prolog and they are retrieved by means of a special primitive (called `clauses`). Although this primitive is defined in Dactl in our implementation, it could be part of a lower-level implementation, perhaps using special hardware. Such an implementation could be very useful in retrieving a large number of facts or rules and it could easily be interfaced to the rest of the Dactl code due to the modular structure of Dactl programs. A complete implementation of PARLOG including the lazy set ('subset') constructor can be found in [**Papa87b**].

3.2 GHC —> Dactl

Suprisingly the flat subset of GHC proved to be very close to Dactl. There are no and-sequential operators or any extralogical features, the language's operational semantics allow everything to be executed in parallel and the required synchronization is defined by the rules of suspension. This is very similar to what is happening in Dactl where everything can be fired in parallel and synchronization is achieved by means of appropriate markings which can directly model the notion of suspension. The only difference between the two languages is the existence of simple guards (in GHC) and an associated *rule of commitment* ([**Ueda86**]):

> — When some clause C called by a goal G succeeds in solving its guard, the clause C tries to be selected for subsequent execution of G. To be selected, C must first confirm that no other clauses in the program have been selected for G. If confirmed, C is selected indivisibly, and the execution of G is said to be committed to the clause C.

The rule of commitment can be easily modelled in Dactl with an `Or_process` that is defined as follows:

```
Or_process[FAIL FAIL] => *FAIL;
Or_process[body:Result[body] Any] => @body;
Or_process[Any body:Result[body]] => @body;
(Or_process[p1 p2]-Or1-Or2) => #Or_process[^p1 ^p2];
```

where

```
Or1=Or_process[FAIL FAIL];
Or2=(Or_process[Result[Any] Any]-Or_process[Any Result[Any]]);
```

There is one `Or_process` for every GHC procedure which fires in parallel all the clauses for that procedure. If it receives back a body it commits to that; otherwise if all the clauses fail it reports failure and terminates. Note that we have adopted here a binary representation; however the technique of using a special module to handle terms of arbitrary size can be used here also.

A clause is represented in essentially the same way as a PARLOG one. However it is the clause itself that is responsible for suspending or failing. For each clause of the form `Pred(Pattern_`

Pattern$_2$ ...) we have three rules:

```
Pred[Pattern1 Pattern2 ...] => *Commit[guard body];
(Pred[p1:(Var+Pattern1) p2:(Var+Pattern2) ...]-P1) => o:#Pred[^p1 ^p2 ...];
(Pred[Any Any ... Any]-P1-P2) => *FAIL;
```

where P1 and P2 are defined in the usual way. Commit is also defined in essentially the same way as in PARLOG. However all guard calls are fired in parallel; no suspension list is built and Commit will eventually respond with either FAIL or the body of the respective clause. Note again the use of a single blocking mark and a number of notification marks. This representation allows the earliest possible detection of failure even if not all input arguments have arrived.

We conclude this section by describing the implementation of the otherwise primitive which can be used to define an 'escape' clause. In particular, the clause that executes an otherwise in its guard will only be tried if all the textually preceding similar (i.e. with the same predicate name and arity) clauses have been tried and failed. Its definition in Dactl is the following:

```
Other_process[Any body:Body] => @body;
Other_process[other ] => @other;
```

An Other_process is called by the top rewrite rule that handles the or-parallelism. The first argument is the clause that calls the otherwise primitive and the second is either a single clause that textually precedes the one with the otherwise or a group of clauses monitored by Or_processes. Other_process remains suspended until all the Or_processes have reported back. If they have all failed, Other_process fires the clause that calls the otherwise primitive.

Appendix II lists a list permutation program that computes all the permutations of the elements of a given list. This program makes use of a novel technique which is described fully in [**Ueda86**] and involves compiling or-parallelism to and-parallelism and using continuations to hold multiple solutions. In particular, a Prolog program is transformed to an equivalent GHC one where the top-down search of the original program has been transformed to a bottom-up one. Provided that the terms involved are ground, the continuations required to hold the multiple solutions do not have to copy the partial solutions that have already been computed which can therefore be shared. Although this method dispenses with the need to create new variants of terms which involves copying, it requires that the terms to be handled are ground. A complete implementation of GHC including some more non-trivial programs is discussed in [**Papa87b**].

4. DISCUSSION

In this section we comment on the techniques used in implementing PARLOG and flat GHC and examine alternative solutions. We then start by noting that the AND tree model of PARLOG can be directly supported by a rewrite rule based language like Dactl. However the existence of a single process which was introduced in the AND tree model to simulate the or-parallelism, compromised to some extent the potential of Dactl for parallel execution and was responsible for the creation of rather lengthy and complicated Dactl code. The reason for the existence of such a process is that the earlier AND/OR tree model proved to create a substantial number of processes, many of which were trivial (for example there would be a single process for each GET-... instruction). This led to an unacceptable overhead when PARLOG programs were compared with similar HOPE programs, both run on ALICE ([**Greg87**]). In Dactl however, there is no need to create processes for the pattern matching instructions and the guards will only contain calls to arithmetic or other predefined system primitives. We therefore propose the following change to the AND tree model: instead of having a single process for a whole procedure, we create one process for every clause (as a whole) of the procedure. That is, a conjunction of calls in a guard is still treated by the same process; however, there is now one process for each clause and all input

matchings and guard evaluations between different clauses proceed in parallel. This technique has some considerable advantages: First, there is no need to build up a list of suspension variables which proved to be the major source of innefficiency in the model. Second, the processes created are not trivial and their number is kept down to manageable levels. Third, some true or-parallelism is supported. We have done some experiments using this technique and the overhead incurred was less than the one using the AND tree model.

A point that needs further explanation is the way we handle failure. Both the AND/OR and the AND tree model assume the existence of a *kill signal* that will catch-up and terminate the processes of an and-conjunction if any of them has failed, or an or-conjunction if one of them has succeeded. However in Dactl there is no such notion of killing processes since such an operation does not have any meaning in a graph reduction model (indeed, it does not have any declarative meaning at all). When such a situation arises we simply overwrite the particular rewrite rule with `SUCCEED` or `FAIL` without considering the rest of its children processes. This situation arises because parallel logic languages (unlike their functional counterparts) have the notion of *speculative work*: processes will be spawned to do work that in the end will prove to be useless. This can happen either because some other process has managed to succeed earlier (as in the case of commited-choice or-parallelism) or because an associated process in the same and-conjunction has failed. The latter reason has also to do with the way the notion of failure is defined. In a functional language if the evaluation of a constraint fails, the result is undefined. In a logic program however, not only failure is valid but is sometimes a way to control the execution of the program. In any case as Ueda points out ([**Ueda86**]) a truly parallel language may have to allow possible useless computation. However the lack of an explicit kill signal in Dactl is usually no problem. Since a process usually needs certain information (input matchings) to proceed, all processes that would otherwise have been killed will eventually stop, suspended on their input arguments and computation will die out.

In the case of GHC, Dactl can be used as the implementation language of such features as the unification primitive which can not be defined in the language itself. In addition, a proper implementation of GHC to Dactl would be beneficial for the language itself since so far only sequential implementations of the language have been reported ([**Ueda86**]) and the potential of the language for efficient parallel implementations is yet to be examined. Although we used a rather liberal technique in implementing GHC (firing everything in parallel), a more conservative one (like the modified AND tree model for PARLOG described above) may prove more appropriate.

We end this section by showing that the use of patterns in Dactl solves some of the problems encountered in implementing head unification ([**Ueda86**]). For example in the following program:

```
goal: :- and(X,false).
clause: and(true,true) :- true | true.
```

the attempted unification should fail and it will in Dactl, whereas in a sequential implementation the call would suspend on the first argument. A more serious problem arises when the following program is encountered:

```
goal: :- p(A,false,true).
clause: p(X,X,X) :- true | true.
```

where a left-to-right unification would suspend whereas a right-to-left one would fail (as it should). We solve this problem by using a cyclic set of equality testing primitives as they are defined in PARLOG. So, the above clause becomes:

```
p(X1,X2,X3) :- X1==X2, X2==X3, X3==X1 | true.
```

5. RELATED WORK AND DIRECTIONS FOR FUTURE RESEARCH

The work reported in **[Hamm86]** examines the potential of Dactl as a compiler target language for functional languages. A lazy subset of ML (SML) has been successfully translated to Dactl and an associated compiler has been built. The paper also examines various compiling techniques and associated problems.

From the point of view of implementing PARLOG we mention here the work reported in **[LaGr87]** where an implementation of the language on the ALICE CTL is examined. The paper also reports on some optimisations that can be applied to the AND tree model. However ALICE CTL is closer to the machine level than Dactl and the programmer is still forced to consider low-level details. We are not aware of any similar work on GHC.

The only other language that comes close to the semantics and aims of Dactl is Lean (**[BEGK87]**) which is also based on graph rewriting. However, the form of graphs that can be defined in Lean is more general (cyclic and even disconnected graphs are allowed). As in the case of Dactl there is no fixed reduction strategy; however, in Lean reduction is specified solely through the patterns of the rewrite rules and there is no use of markings to nominate certain nodes as potential redexes.

The use of Dactl as an intermediate compiler target language provides the ability to reason about the performance of various logic and functional languages. In particular, mapping programs written in different languages onto Dactl allows us to examine and compare the resulting code. Such a comparison can be done for example between programs written in PARLOG or GHC and ML. We can also experiment on various implementation models for the languages themselves.

Finally, we observe that in Dactl there is no distinction between data constructors, predicates and functions. This reveals the possibility of using Dactl as the basis for amalgamating logic and functional languages. This may be done in 2 ways: First, Dactl can be used to implement languages like FUNLOG (**[SuYo86]**) based on *semantic unification*. In semantic unification the most general unifier is unique and the arguments of the functions to be reduced must be ground terms. Thus there is no need for either backtracking or creation of local environments. We believe that Dactl can efficiently support the implementation of languages like FUNLOG. Second, the very fact that both logic and functional languages can be mapped onto Dactl may provide a means of either interfacing these languages to each other or implementing in Dactl 'expensive' operations such as *narrowing* (**[Redd86]**) which may be called when necessary by the languages concerned.

6. CONCLUSIONS

In this report we have examined the potential of Dactl as an intermediate compiler target language for parallel logic languages. In particular we provided a framework for compiling PARLOG and flat GHC programs in which all the features supported by these languages (stream parallelism, back unification, lazyness, continuations, or-parallelism *a la* PARLOG, bounded buffer communication etc.) can be easily and efficiently implemented in Dactl. We also showed that all the control or data driven annotations used in these languages are either directly supported or efficiently implementable in Dactl. In general, we believe that Dactl can be used as the compiler target language for a wide class of parallel logic languages that conforms to the following two principles:

- There is no need for creating local environments during the search for candidate clauses.
- The input/output directionality of programs is explicit at compile time, that is the programs should be *moded*.

We impose the first restriction so that we can dispense with the need to build copies of local variables and subsequent merging of local and global environments which has been proved to be problematical (**[Ueda85]**). The second restriction is imposed so that we can produce the required Dactl code at compile time without having to resort to run-time checks (as for example is the case of

full GHC). However we stress the fact that input arguments need not be *strong* (where only ground terms are allowed). The use of *weak* arguments (where only the outermost symbol is defined) can be easily supported in our model. Although even at this stage we are able to offer a limited degree of unification we believe that if we allow some form of copying we will be able to accomodate a wider class of languages such as Concurrent Prolog ([**Shap83**]) or P-Prolog ([**YaAi86**]) and execution models such as narrowing ([**Redd86**]).

7. REFERENCES

[BEGK87] Barendregt H. P., van 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,
Internal Report SYS-C87-02, University of East Anglia, U.K., also PARLE Conference, Eidhoven, Netherlands, June 15-19.

[ClGr85] Clark K. and Gregory S.,
Notes on the Implementation of PARLOG.,
Journal of Logic Programming 2, 1, pp. 17-42.

[Ders85] Dershowitz N.,
Computing with Rewrite Rules,
Information and Control 65, 122-157.

[GKSH87] Glauert J. R. W., Kennaway J. R., Sleep M. R., Holt N. P., Reeve M. J. and Watson I.,
Specification of Core Dactl1,
Internal Report SYS-C87-09, University of East Anglia, U. K.

[Greg87] Gregory S.,
Parallel Logic Programming in PARLOG: the Language and its Implementation.
Addison-Wesley, ISBN 0-201-19241-1.

[Hamm86] Hammond K.,
Compiling ML to Dactl: Early Experience,
Internal Report SYS-C86-04, University of East Anglia, U. K.

[HuOp80] Huet G. and Oppen D. C.,
Equations and Rewrite Rules: a Survey,
Technical Report CSL-111, SRI International.

[LaGr87] Lam M. and Gregory S.,
PARLOG and ALICE: a Marriage of Convenience,
4th International Conference on Logic Programming, University of Melbourne, Australia, May 25-29.

[Papa87a] Papadopoulos G. A.,
Parallel Execution of Logic Programs: a Survey.
Internal Report SYS-C87-08, University of East Anglia, U.K.

[Papa87b] Papadopoulos G. A.,
Towards a Common Implementation for Parallel Logic Languages Based on an Intermediate Compiler Target Language.
Internal Report (to appear), University of East Anglia, U.K.

[Redd86] Reddy U. S.,
Graph Narrowing of Functional Logic Languages,
Graph Reduction Workshop, Los Alamos National Laboratory, U.S.A., Sept. 29-Oct. 1.

[Shap83] Shapiro E. Y.,
A Subset of Concurrent Prolog and its Interpreter,
ICOT, Japan, TR-003.

[SuYo86] Subrahmanyam P. A. and You J. -H.,
FUNLOG: a Computational Model Integrating Logic Programming and Functional Programming,
in Logic Programming: Functions, Relations and Equations, Prentice-Hall, ISBN 0-13-539958-0, pp. 157-198.

[Ueda85] Ueda K.,
Concurrent Prolog Re-examined,
ICOT, Japan, TR-102.

[Ueda86] Ueda K.,
Guarded Horn Clauses,
D.Eng. Thesis, University of Tokyo, Japan.

[YaAi86] Yang R. and Aiso H.,
P-Prolog: a Parallel Logic Language Based on Exclusive Relation,
3rd International Conference on Logic Programming, Imperial College, London, U. K., July 14-18, LNCS 225, Springer-Verlag, pp. 255-269.