PARALLEL IMPLEMENTATIONS OF DECLARATIVE LANGUAGES BASED ON GRAPH REWRITING

K. Hammond and G.A. Papadopoulos

University of East Anglia, UK

0. INTRODUCTION

Dactl is a compiler target language based on a parallel graph rewriting model of computation. Throughout this paper, the reader is assumed to be familiar with the syntax and semantics of Dactl as presented in e.g. Glauert (1). In particular, it is necessary to thoroughly understand Dactl control markings, patterns and rule-ordering. More esoteric points of Dactl usage will be explained carefully where appropriate.

The reader is also assumed to possess at least a working knowledge of functional and logic programming languages. Examples in this paper use Standard ML, henceforth SML, (Milner et al. (2)); PARLOG (Gregory (3)); and safe GHC (Ueda (4)).

SML was chosen as an example of a functional language which is widely used, with many interesting (difficult to compile) features. This paper considers only a purely functional subset of SML, Hammond (5) treats the translation of the non-declarative features.

PARLOG and GHC were chosen as representative of the concurrent logic family of languages because they are "safe"; any computation performed in the "passive" part of a clause (head and guard) is allowed only to test the values of variables. The implementation of the run-time safety check for GHC is described by Glauert and Papadopoulos (6). Certain advanced PARLOG metaprogramming features are discussed in detail by Papadopoulos (7).

1. FUNDAMENTAL TECHNIQUES

To illustrate the basic features of our techniques, we initially restrict ourselves to minimal subsets of the languages which concern us. For functional languages, we consider first-order functions without nested definitions (let or where) and with flat patterns. For logic we consider unguarded clauses with non-overlapping patterns.

It is necessary to consider functional and logic languages independently, due to the different computational models adopted. The evaluation of a function application will deliver a single result, whilst the evaluation of a logic clause will simply succeed or fail, results being obtained through instantiation of logic variables. Whichever model of computation is chosen, the basic form of a source program is equivalent: a set of rules defining the behaviour of a rule-system; a goal expression defining the problem to be solved using this rule-system and a set of predefined *delta* rules defining micro-operations such as arithmetic, conditionals etc.

The target program will have a similar form: a single Dactl module comprising several rule-groups (one for each source rule-group) with their associated declarations; a single rule for the special pattern *Initial*; and a list of imported modules which define the necessary delta-rules. In practice, many of these imported modules may be implemented directly on a given machine rather than as Dactl code. All ground forms will be nodes whose symbol has the *Creatable* access class, whether predefined Dactl types such as *Int.* etc. or user-defined constructors.

1.1. Functional

The basic principle of implementing a functional language is that every expression should, when evaluated, reduce to the normal form of that expression, if one exists. (For a lazily evaluated language the condition can be rather weaker: the goal expression must reduce to normal form, and other needed expressions must reduce to weak-head normal form.) For the time being we will consider only a fully strict implementation, thus we require that on activation any translated SML expression will reduce to its normal form. Synchronisation is achieved where necessary through Dactl notification, utilising the fact that no rules will exist for expressions which are in a normal form.

One of the simplest possible functions is the identity function:

```
fun I x = x;
```

This can be trivially translated into the Dactl rule (we will ignore symbol declarations):

```
I [x] => *x;
```

It is necessary to activate *x* in order to cause notification. In this translation function application is mapped into a node whose symbol is the functor and whose children are the arguments to the function. Activation of such a node will then result in the reduction of the function application. For instance:

```
fun    Comb x y = I (x + y);

=>     Comb[x y] => #I[^*ADD[x y]];
```

There are several points to notice about this translation. First is the treatment of the multiple arguments to *Comb*. Second is the suspension of the application of *I* pending the evaluation of its argument. Last is the use of a delta-rule *ADD* to perform addition. The Dactl special rules for addition, *IAdd* etc. cannot be used directly since *x* or *y* may be either integers or reals. Use of typechecking information would allow the direct use of the special rules, but at the potential expense of compiling four rules for *Comb*. Such a translation would also cause complications if exception handling and lazy evaluation were permitted. With a fully strict implementation, *x* and *y* must already be in normal form, therefore there is no need to activate them before performing the addition.

So far the rules given have used only simple variables as arguments. It is easy to extend the translation scheme to more general patterns, however. Translating the ubitiquous naïve Fibonacci function:

```
fun Fib 0 = 1
|   Fib 1 = 1
|   Fib n = Fib (n-1) + Fib (n-2);

=>  Fib[0] => *1;
    Fib[1] => *1;
    Fib[n:Int] => ##ADD[^#Fib[^*SUB[n 1]]
                        ^#Fib[^*SUB[n 2]]];
```

The first two rules translate the pattern directly, the third tags the variable with its type, strictly unnecessary in this case. Note the use of rule sequencing, the third rule will only match integers which are neither 0 or 1. This sequencing proves especially useful when translating complex patterns with lazy evaluation. In the third rule, we see parallelism for the first time: computations of *fib (n-1)* and *fib (n-2)* are free to proceed in parallel. This parallelism is implicit, derived only from the information that all functions are strict. The parallelism is in this case probably superfluous since there exist far better sequential algorithms for the calculation of Fibonacci numbers, but it does illustrate the triviality of deriving parallelism from "Divide and Conquer" algorithms.

Such a translation is readily extended to functions of several arguments and those using user-defined constructor functions. Note the direct translation of constructor functions into Dactl symbols.

```
datatype Logic = On | Off;
datatype States = S1 of Logic | S2 of Logic;
fun    state (S1 s) On  _  = S2 s
|      state (S1 _) _ _  = S1 On
|      state _ s Off = S1 s
|      state (S2 _) _ _ = S2 Off;

=>     State[S1[s] On Any] => *S2[s];
       State[S1[Any] Any Any] => *S1[On];
       State[Any s Off] => *S1[s];
       State[S2[Any] Any Any] => *S2[Off];
```

The first real complications occur when lazy evaluation is introduced. Lazy evaluation is desirable because of the flexibility it provides the functional programmer. However, by its nature lazy evaluation imposes sequence on otherwise parallelisable sections of code, which is obviously undesirable. We will treat this issue later.

Since it is no longer possible to assume that all arguments have been reduced to normal form, or even head normal form when they are passed to a function, it is necessary to force evaluation where necessary. We assume that there exist versions of *ADD, SUB* etc. which do this, called *ADDL, SUBL* etc. Then the translation of the naïve Fibonnaci function given above will become:

```
Fib[0] => *1;
Fib[1] => *1;
Fib[n:Int] => *ADDL[Fib[SUBL[n 1]]
                    Fib[SUBL[n 2]]];
Fib[n] => #Fib[^*n];
```

The pattern tag, *int,* on the third rule is now necessary since the argument to *Fib* could be an unevaluated expression. Such an expression must be reduced to head normal form in case it matches either of the literal patterns. If, however, an SML pattern is a variable or the wildcard pattern then it may not be necessary to force evaluation. As illustration consider the following translation of the Ackermann function:

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

=>     Ack[0 n] => *ADDL[n 1];
       Ack[m:Int 0] => *Ack[SUBL[m 1] 1];
       Ack[m:Int n:Int] =>
            *Ack[SUBL[m 1] Ack[m SUBL[n 1]]];
       Ack[m:(Any-Int) n] => #Ack[^*m n];
       Ack[m n] => #Ack[m ^*n];
```

The first rule does not require the value of *n* in order to match the pattern (though it will be necessary to perform the addition). However, the value of *n* is required if the second and third rules are to be differentiated. Two default rules are required to force evaluation of either argument. Note that priority is given to the first argument, in accordance with the standard semantics for pattern-matching in a lazy functional language.

SML control statements (IF, ANDALSO, ORELSE etc.) are necessarily non-strict, and are treated identically to lazy delta-rules. Note that ANDALSO, ORELSE in particular are sequential operators, it is a simple matter to write parallel versions if required. To illustrate this we give the Dactl rules for ANDALSO. For a strict implementation the default rule would be omitted and the appropriate arguments activated in place.

```
ANDALSO[TRUE c2] => *c2;
ANDALSO[FALSE c2] => *FALSE;
ANDALSO[c1 c2] => #ANDALSO[^*c1 c2];
```

In fact, these lazy control structures cause complications in a strict translation since it is not possible to annotate the successors of these nodes. The solution is to use source-to-source transformation to introduce new functions to replace the unevaluated portions of the control structure. Since this generally introduces some overhead through chaining of parameters, it is obviously pointless to compile out simple expressions.

1.2 Logic

Consider again the Fibonacci function, this time written as a GHC predicate:

```
fibonacci(N) :- true | fib(1,0,N).
fib(N1,N2,Ns0) :- true | N3:=N1+N2, Ns0=[N3|Ns1],
                         fib(N2,N3,Ns1).
```

This is translated to Dactl as follows:

```
Fibonacci[n] => *Fib[1 0 n];
Fib[n1 n2 ns0] => #AND[^b1 ^b2 ^b3],
                  b1:#Evall[n3:Var ^*Plus[n1 n2]],
                  b2:*Unify[ns0 CONS[n3 ns1:Var]],
                  b3:*Fib[n2 n3 ns1];
```

Note that in a concurrent logic language, unless otherwise indicated (by means of sequential operators or special calls), everything is performed in parallel: head unifications, attempts to commit to a clause, and conjunctive goals within guards and bodies are all evaluated in parallel. In the case of GHC, the body calls can commence execution even before commitment and in parallel with their respective guards. In the above example, the three calls in the body of *Fib* are executed in parallel and are monitored by an *AND* process which is defined as follows:

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

AND monitors the execution of its child processes; if they all complete sucessfully it reports success and terminates (first rule). Otherwise, if any of them fails it reports failure and also terminates (last rule). The use of a single *#* causes the *AND* process to be activated every time it receives a signal from any of its children. Thus if some children have reported successful termination but others are still executing, *AND* is re-suspended awaiting the outcome of their computation (second rule). In this way the earliest possible detection of failure is achieved. Note here that unlike a functional language, failure is a valid result in a logic language which may, in fact, cause the creation of additional useful computation. Finally, note that any new variables on the right hand side of a clause are represented in Dactl as new nodes with the pattern *Var.*

In the above example we do not need to consider suspension on uninstantiated input arguments since all the head arguments are variable terms. However, this is not always the case as it is illustrated by the following *merge* program written in PARLOG and its translation to Dactl:

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

merge([A|X],Y,[A|Z]) <- merge(X,Y,Z).
merge(X,[A|Y],[A|Z]) <- merge(X,Y,Z).
merge([],Y,Y).
merge(X,[],X).

Merge[CONS[a x] y z1] => #AND[^b1 ^b2],
                         b1:*Unify[z1 CONS[a z:Var]],
                         b2:*Merge[x y z]|
Merge[x CONS[a y] z1] => #AND[^b1 ^b2],
                         b1:*Unify[z1 CONS[a z:Var]],
                         b2:*Merge[x y z]|
Merge[Nil y z] => *Unify[z y]|
Merge[x Nil z] => *Unify[z x];
(Merge[p1 p2 p3]&
(Merge[(Var+CONS[Any Any]) Any Any]+
 Merge[Any (Var+CONS[Any Any]) Any]))
                               => #Merge[^p1 ^p2 p3];
Merge[Any Any Any] => *FAIL;
```

If the input arguments of *merge* are not sufficiently instantiated, the fifth rule is selected causing suspension of the computation until the required data have arrived (note here the use of Dactl pattern operators to detect whether to suspend or fail). If these arguments are eventually instantiated to something other than a list, the last rule will be selected to report failure. Note also that if the body of a clause comprises a single call, there is no need for an *AND* process and a more direct representation is possible as illustrated by the third and fourth rules.

2. IMPROVEMENTS TO THE BASIC SCHEMES

Having discussed the principles of the translation process, we now consider improvements which are essential in order to compile real functional or logic programs, or desirable in terms of efficiency. We restrict our attention to declarative constructs only.

2.1. Functional

The most important omission from the translation scheme presented earlier is that of higher-order functions. These present problems through the use of general function application, where the function applied is in fact an expression which reduces to a function, and also through the use of Currying. Considering general function application first, a natural representation is through a binary function application operator *AP*, whose first operand is an expression representing a function, and whose second operand is an expression representing the function argument. Using the Dactl symbol representing a function in a nullary node allows functions to be manipulated as values in the correct manner: since no rules are provided for these forms such nodes act like constructors.

Obviously, normal functions and delta rules must be translated in such a way that they may be used in conjunction with *AP* as higher-order functions. This can be achieved by adding rules for each function for each case where the function has insufficient arguments. For example (lazy version):

```
fun  partition x [] lt gt = (lt, gt)
|    partition x (y::ys) lt gt=
        if y < x then partition x ys (y::lt) gt
                 else partition x ys lt (y::gt);

=>   Partition[x NULL lt gt] => *TUPLE[lt gt];
     Partition[x CONS[y ys]lt gt]=>
       *IF[LTL[y x] Partition[x ys CONS[y lt]gt]
                    Partition[x ys lt CONS[y gt]]];

     AP[Partition x] => *Partition[x];
     AP[Partition[x] ys] => *Partition[x ys];
     ...............
```

Partial applications may be translated into either the functional style used for total applications if the function is a function name, or into the applicative style using explicit *AP*s.

Now, of course, it is no longer true that activating any node will cause reduction to head normal form for a lazy implementation: no rules exist to reduce *AP* nodes where the function part is a general expression or a sequence of *AP* nodes. This can be remedied by supplying a default rule for *AP*, but there is a complication: Dactl does not allow us to specify sequence between rule-groups, they may be tested in parallel. A simple solution is to supply all the rules for *AP* (including those acting on delta-rules) in one rule-group. Then the final rule in the group is simply:

```
AP[f x] => #AP[^*f x];
```

So far we have considered only the reduction to head normal form of a lazily evaluated program. Often this is what is required, since normal form and head normal form are the same for nullary constructors. The following Dactl rules force reduction to normal form of any node, and can therefore be used in the *Initial* node. Such rules, are of course unnecessary for a strict evaluation strategy.

```
RNF[e] => #RNF'[^*x];

RNF'[e] => #n (symbol (e))[^*RNF[(successor (e, 1)]
              ........ ^* RNF[(successor (e, n)]];
```

The rule for *RNF'* is encoded in Dactl using "screwdriver" operations to manipulate the structure of its argument.

A similar set of rules, though more detailed in order to account for differing output formats, is used in the translator to print the result of a lazy program. This permits the execution of lazy programs generating infinite data structures or the viewing of results as they are produced.

Deep patterns may be handled either directly, in which case multiple default rules are required to evaluate unreduced, needed expressions at each point in the pattern, or by transformation of the source into a simpler form containing only single level patterns. The latter is probably the better approach, but requires some compilation effort to coalesce the source patterns, and introduces overhead in the form of needed arguments carried forward. For example:

```
fun first (x1::x2::xs) = x1;
=>
fun first (x1::rest) = first' x1 rest
and first' x1 (x2::xs) = x1;
```

In conjunction with strictness analysis this approach eliminates the requirement for default rules in the Dactl code. This observation is exploited in a translator from the functional language Clean to Dactl described by Kennaway (8).

Since SML is a block-structured language whereas Dactl is "flat", we use lambda-lifting on the SML source before performing the transformation. It is necessary at this stage to prevent identifier name clashes. This is achieved in the prototype SML compiler through "colouring" each identifier uniquely.

To improve the space efficiency of the implementation, common sub-expressions and maximal free expressions may be eliminated. Such eliminations naturally take advantage of the graph rewriting abilities of Dactl. Common sub-expression elimination is straightforward: maximal free expressions present certain complications, however, since these must be shared between different activations of the same rule in order to achieve full laziness. Unfortunately, Dactl has no direct mechanism to express globally accessible nodes. A possible solution is to pass these expressions as parameters of each rule activation, but such overhead seems unacceptable in general.

Strictness analysis is used to determine which arguments to a function may be evaluated in parallel for a lazy scheme. This is important because strictness is the only source of parallelism in a functional language. As with the non-strict control structures, unevaluated arguments which could be annotated with strictness information must be compiled into separate rules. For the prototype SML compiler, we use only the simple technique for strictness analysis described by Peyton-Jones (9).

2.2. Logic

In section 1.2 we showed how programs without guards may be translated into Dactl. In this section we extend this method to handle guarded clauses. Consider the *partition* program rewritten in GHC and its translation to Dactl:

```
partition([X|Xs],A,S,L0) :- A<X | L0=[X|L1],
                             partition(Xs,A,S,L1).
partition([X|Xs],A,S0,L) :- A>=X | S0=[X|S1],
                             partition(Xs,A,S1,L).
partition([],_,S,L) :- true | S=[], L=[].

Partition[CONS[x xs] a s l]
         => #Partition_Commit[^g1 ^g2 x xs a s l],
                   g1:*Less[a x], g2:*Lesseq[a x]|
Partition[Nil Any s l] => #AND[^b1 ^b2],
                          b1:*Unify[s Nil],
                          b2:*Unify[l Nil]]|
Partition[p1:Var p2 p3 p4]
                       => #Partition[^p1 p2 p3 p4];
Partition[Any Any Any] => *FAIL;

Partition_Commit[SUCCEED Any x xs a s l0]
                    => #AND[^b1 ^b2],
                       b1:*Unify[l0 CONS[x l1:Var]],
                       b2:*Partition[xs a s l1]|
Partition_Commit[Any SUCCEED x xs a s0 l]
                    => #AND[^b1 ^b2],
                       b1:*Unify[s0 CONS[x s1:Var]],
                       b2:*Partition[xs a s1 l]|
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 input patterns, they can be coalesced into a single Dactl rule which performs the required input matching once only. We are then left with two non-overlapping rules; the first evaluates the two guards and commits to the body of either the first or the second clause; the second closes the output streams if its first argument is the empty list. In general, guards are evaluated by a process which takes the form:

```
Predname_Commit[guards head_and_guard_vars]
```

where *guards* is a set of processes, one process for each guard conjunction, and *head_and_guard_vars* is the set of variables appearing in the head of the clause plus any new variables appearing in the guards. Thus the environments of the head and guard evaluations are carried forward and used when *Predname_Commit* commits to the appropriate body.

Certain guarded clauses possess overlapping input patterns which cannot be coerced to a set of rules with non-overlapping patterns as shown above. For such clauses we proceed as follows: since all clauses in the procedure must be tried in parallel, we first transform the overlapping patterns into non-overlapping ones by introducing a new function symbol for each set of overlapping patterns; then we fire all rules in parallel using a Dactl "metarule". Consider the following program fragment which is part of a GHC meta-interpreter (where we assume that the guard in the second clause is safe) and its translation to Dactl:

```
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).

Pred[p1 p2 p3] => #Search[^#OR[^o1 ^o2]],
                  o1:*Pred1[p1 p2 p3],
                  o2:*Pred2[p1 p2 p3];

Pred1[True x Any] => *Result[Unify[x "Ok"]]|
Pred1[Cl[a b] x c] => *Result[Body[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]|
Pred1[p1:Var p2 p3] => #Pred1[^p1 p2 p3]|;
Pred1[Any Any Any] => *FAIL;

Pred2[a x Any] => #Pred2_Commit[^g a x],
                  g:*Ghcsystem[a];

Pred2_Commit[SUCCEED a x]
                         => *Result[Body[[b1 b2]],
                            b1:Unify[x "Ok"],
                            b2:Call[a]|
Pred2_Commit[FAIL Any Any] => *FAIL;
```

We now show how we handle deep pattern matching in the case of logic languages. The problem with deep patterns is that many processes may be working on the same data structure, each instantiating its own part. It is difficult to know at any given time how much of the data structure has been created and how much is yet to be constructed. This knowledge is essential in order to suspend on those parts of the structure which are needed but are still uninstantiated. Consider the following PARLOG program and a possible translation to Dactl:

```
mode m(?).

m([f(g(X))|T]).

M[CONS[TUP["F" TUP["G" x]] t]] => *SUCCEED|
M[p:Var] => #M[^p]|
M[CONS[h:Var t]] => #M[^#CONS[^h t]]|
M[CONS[TUP["F" x:Var] t]]
                          => #M[^#CONS[^#TUP["F" ^x] t]];
M[Any] => *FAIL;
```

Unfortunately, the rules needed to model suspension must cover all possible combinations of input patterns and therefore their number can be unnacceptably high. An alternative method where we compile the required pattern matching into a set of primitive pattern matches, each handled by a different process is described in (7).

Finally, we decribe the implementation of the 3-argument metacall as defined in (3). This is used for systems programming and metaprogramming. A call of the form *call(p,s,c)* is executed as follows: if *p* completes execution successfully, *call* instantiates its status variable *s* to *SUCCEED* and terminates. If *p* fails, *call* instantiates *s* to *FAIL* and also terminates. If, during the evaluation of *p*, *call* receives a *STOP* message through its control argument *c*, it kills the process *p* and terminates. If *call* is called with its first argument instantiated to something other than a valid predicate, it fails. A possible implementation in Dactl is shown below:

```
Call[Any s:Var STOP] => *SUCCEED, s:=*STOP|
Call[p:Var s:Var c:Var] => #Call[^p s ^c]|
Call[p:(SUCCEED+FAIL) s:Var Var] => *SUCCEED,
                                    s:=*p|
Call[p:Valid_Pred s:Var c:Var] => #Call[^*p s ^c];
Call[Any Any Any] => *FAIL;
```

The propagation of the *STOP* signal down the computation tree is achieved by extending the rule system for *p* with an additional rule which checks whether an extra, special argument has been instantiated to *STOP*. Consider the following:

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

append([U|X],Y,[U|Z]) <- append(X,Y,Z).
append([],Y,Y).

Append[STOP Any Any Any] => STOP|
Append[s:Var x:Var y z] => #Append[^s ^x y z]|
Append[s:Var CONS[u x] y z:Var]
                               => *Append[x y z1],
                                  z:=*CONS[u z1:Var]|
Append[s:Var Nil y z:Var] => *SUCCEED, z:=*y;
Append[Any Any Any Any] => *FAIL;
```

3. Suitability of Dactl as a CTL

With the exception of global nodes which are needed to allow efficient maximal free expression elimination, Dactl provides a sound framework for the translation of pure functional languages, focussing attention on the compilation issues involved rather than on machine issues. All purely functional constructs may be translated. The lack of fine control over machine resource allocation is somewhat disconcerting, though necessary if Dactl is not to impose constraints on the architectures which support it. Machine-dependent annotations will, however, permit some fine-tuning where the target machine characteristics are known. For functional languages, deep pattern-matching presents certain implementation problems, as do general function application and the equality test (for similar reasons).

As far as safe concurrent logic languages are concerned, we have shown that all their features can be implemented in Dactl. Some of them, however, need to be programmed around and lead to less efficient code. Guarded clauses with overlapping patterns, for example, must be fired in parallel by means of an extra metarule which, in addition, will monitor their progress. This is necessary because Dactl is a rewrite-rule based language with no notion of backtracking. Deep pattern matching causes the creation of rather complicated code, as was also the case for functional languages. Finally, the killing of unnecessary computation, although it can be achieved in principle by means of the transformation technique shown in the previous section, is not always effective. Since the speed of propagation of the kill signal depends on the scheduling of processes, it can lead to race conditions. What is needed here is a low-level implementation which, however, is not easy to develop in a parallel graph reduction model: it is hard to detect unneeded but active portions of graph. Note that the root of the problem stems from the existence of speculative parallelism in the concurrent logic model which, in general, is difficult to control on a parallel architecture.

4. Conclusions

We have demonstrated the feasibility of translating several declarative languages into the parallel intermediate language Dactl, highlighting areas of especial difficulty. The fundamental model of parallel graph reduction supported by Dactl has thus been shown to be sufficiently expressive to accommodate both functional and logic styles of programming. A prototype compiler for SML, based on the techniques illustrated here, has been written in C. A similar compiler for GHC is currently under construction, built on top of the SPM (PARLOG) system. Since implementations of Dactl are being produced for a variety of parallel machines (Flagship, GRIP, Meiko Transputer Rack etc.), these compilers will enable these languages to execute in a real parallel environment.

References

1. Glauert, J.R.W., 1988, "An Introduction to Graph Rewriting in Dactl", Proc. Alvey Technical Conference.
2. Milner, R., 1984, "The Standard ML Core Language", Edinburgh University, Internal Report, CSR-168-84.
3. Gregory, S., 1987, "Parallel Logic Programming in PARLOG: the Language and its Implementation", Addison-Wesley, London.
4. Ueda, K., 1986, "Guarded Horn Clauses", D.Eng. Thesis, University of Tokyo, Japan.
5. Hammond, K., 1988, "Implementing Functional Languages on Parallel Machines", Ph.D. Thesis, University of East Anglia, in preparation.
6. Glauert, J.R.W., and Papadopoulos, G.A., 1988, "A Parallel Implementation of GHC", submitted for publication.
7. Papadopoulos, G.A., 1988, "Compiling PARLOG into Dactl", University of East Anglia, Internal Report, (to appear).
8. Kennaway, J.R., 1988, "Implementing Term Rewrite Languages in Dactl", Proc. CAAP '88, Lecture Notes in Computer Science, 299, 102–116, Springer-Verlag, Berlin.
9. Peyton-Jones, S.L., 1987, "The Implementation of Functional Programming Languages", Prentice-Hall, London.