PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT. © ICOT, 1988

A PARALLEL IMPLEMENTATION OF GHC

John R. W. Glauert and George A. Papádopoulos

Declarative Systems Project,
University of East Anglia, Norwich NR4 7TJ, U.K.
{jrwg,gp}@sys.uea.ac.uk

ABSTRACT

A parallel implementation of Guarded Horn Clauses (GHC) using graph rewriting techniques is described. GHC programs are mapped to rewriting rules in Dactl, a compiler target language based on generalised graph rewriting. We provide a complete translation scheme for unrestricted GHC programs paying particular attention to the OR-parallelism involved and show that the run-time test required in the presence of user defined calls in the guards can be easily and efficiently implemented in our model. We show that our model supports a variety of execution schemes: evaluating the body of a clause after commitment to that clause or in parallel with the evaluation of the guard. We provide a number of examples throughout to illustrate our techniques.

1 INTRODUCTION

GHC (Ueda 1986) belongs to the class of the so called *committed-choice non-deterministic logic languages*; other members of this class are PARLOG (Gregory 1987) and Concurrent Prolog (Shapiro 1983). These languages support committed-choice OR-parallelism and stream AND-parallelism. In this paper we describe a parallel implementation of GHC based on *graph rewriting*.

According to a graph rewriting model of computation, programs are represented as labelled directed graphs. A set of rewriting rules describes transformations which may be performed on the graph by identifying and rewriting reducible expressions, or *redexes* (Barendregt *et al.*). Independent redexes may be rewritten in any order, or concurrently, so graph rewriting provides a natural model for expressing parallel computations in which a number of processors may cooperate to rewrite a single graph with no need for centralised control.

We show that GHC programs can be viewed as sets of rewrite rules to be applied to an initial graph corresponding to a goal query. On rewriting redexes within this graph until none remain, a final stage will be reached where the resulting graph represents the answer to the query. GHC clauses are translated to Dactl (Glauert *et al.* 1987), a compiler target language based on graph rewriting intended to serve as an interface between new generation declarative languages and novel computer architectures.

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.

2 DACTL

A program in Dactl (Declarative Alvey Compiler Target Language) 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 (Kennaway 1988), the translation of GHC 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 no predefined strategy is adequate, Dactl employs *control markings* to determine the order of reduction.

As an example, the following fragment of Dactl defines some rules for an `Append` function:

```
RULE
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.

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 new graph structure which may contain references to the pattern graph. After a successful matching, a copy of the contractum is built, adding 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:

```
RULE
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 very 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 GHC 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.

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. The following fragment is not from the GHC translation, but illustrates the principles using a logic programming version of `Append`:

```
RULE
Append[x:Var y v] => #Append[^x y v]|
Append[Nil y v:Var] => *Succ, v:=*y|
Append[Cons[h t] y v:Var] => *Append[t y n:Var],
                             v:=*Cons[h n];
```

Note the presence of the non-root overwrites in the rhs of the last two rules using ':=' to instantiate the third (output) argument of `Append` to the appropriate value.

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.

Finally, note that repeated identifiers in the pattern of Dactl rules are allowed, and they are taken to denote a test for pointer equality during matching. In the next section we will see that in the graph rewriting framework as supported by Dactl, the pointer equality test is all that is needed to implement GHC's run-time test.

3 GHC

We assume familiarity of the reader with GHC; here it suffices to mention only the rules of suspension and the rule of commitment:

- Unification invoked directly or indirectly in the guard of a clause C called by a goal G cannot instantiate the goal G.
- Unification invoked directly or indirectly in the body of a clause C called by a goal G cannot instantiate the guard of the clause C or the goal G until C is selected for commitment.
- When some clause C called by a goal G succeeds in solving its guard, it tries to be selected for commitment. To be selected, C must first confirm that no other clause in the program has been selected for G. If confirmed, C is selected indivisibly.

4 IMPLEMENTATION

The first part of this section describes the implementation of the flat subset of GHC; we then go on to extend this basic framework to accomodate the full version of the language where calls to user defined predicates are allowed in the guards. The translation to Dactl illustrated here is simplified, but captures the essence of our model.

4.1 Flat GHC

Here we assume that head unification, guard evaluation and body evaluation (after commitment) are performed in that order. In addition, repeated occurrences of variables in the head are not allowed; these are eliminated by means of extra calls to the unification primitive in the guards. We then start by noting that the first rule of suspension (see previous section) suspends any head unification that attempts to instantiate a variable in the call to a non-variable term in the head; in other words all the head arguments that have non-variable patterns specify conditions that must be satisfied by input data received from the call. There is therefore an implicit input-output *moding* of the clauses as in the case of PARLOG. In fact, a clause does one of three things: it evaluates the guard if the required input patterns have been produced, it fails if the produced input patterns are incompatible with the required ones, and it suspends otherwise. We therefore translate any GHC clause to three Dactl rewrite rules that model success, failure and suspension of head unification respectively. The GHC program

```
ex([H|T],x) :- g1(H) | b1(T,X,Y).
ex(X,f(Y)) :- g2a(Y), g2b(Z) | b2a(X), b2b(Z).
```

may be translated to Dactl as follows:

```
{0}  Ex[p1 p2] => #Search[^#OR[^o1 ^o2]],
                  o1:*Ex1[p1 p2], o2:*Ex2[p1 p2];
{1a} Ex1[Cons[h t] x]
                      => #Ex1_Commit[^*G1[h] t x];
{2a} Ex1[p1:Var p2] => #Ex1[^p1 p2];
{3a} Ex1[Any Any] => *FAIL;
{4a} Ex1_Commit[SUCCEED t x] =>
                            *Result[B1[t x y:Var]];
{5a} Ex1_Commit[FAIL Any Any] => *FAIL;
{1b} Ex2[x Tup["F" y]]
                       => #Ex2_Commit[^guard x z],
                guard:#AND[^*G2a[y] ^*G2b[z:Var]];
{2b} Ex2[p1 p2:Var] => #Ex2[p1 ^p2];
{3b} Ex2[Any Any] => *FAIL;
{4b} Ex2_Commit[SUCCEED x z] =>
                       *Result[Body[B2a[x] B2b[z]]];
{5b} Ex2_Commit[FAIL Any Any] => *FAIL;
```

The top level rule `{0}` activates a set of parallel computations, one to evaluate the guard of each clause of the GHC predicate. Each guard computation will either fail, succeed, or suspend awaiting instatiation of goal variables. Clauses whose guards succeed must not proceed to evaluate the corresponding body goals immediately, since there may be multiple successful guards. Successful guards therefore build a closure `Result[body]`, where `Result` is a constructor, and the argument `body` will evaluate the appropriate body if activated.

The `Search` node and the tree of suspended OR nodes ensure committment of no more than one clause. The OR nodes will be notified on completion of the guard computations, becoming active when the first notification signal arrives since there is only one suspension marker. As soon as one guard succeeds, the OR node propagates the closure for the corresponding body up the tree to the `Search` node. If failure reaches the `Search` node then all guards have failed and `Search` reports failure of the goal. The definition of `Search` and OR is given by:

```
OR[FAIL FAIL] => *FAIL|
OR[res:Result[Any] Any] => *res|
OR[Any res:Result[Any]] => *res;
OR[FAIL p] => p|
OR[p FAIL] => p;
Search[p:FAIL] => *p|
Search[Result[body]] => *body;
```

The computation for the guard of each clause uses a separate set of three rules: The first rule `{1a,b}` models successful head unification when all the required patterns are available. The function then attempts to solve the corresponding guard by activating a Dactl graph of the form:

```
#Clausename_Commit[^guard_conj env]
```

where `guard_conj` is either a single call or a conjunction of such calls as described below. `env` is the set of variables imported by the body from the head or guard. New GHC variables in the guard or body of a clause appear as new Dactl nodes with the pattern `Var`. If `Clausename_Commit` succeeds in solving its guard, it overwrites itself to the code for the respective clause body which is wrapped in the constructor `Result {4a,b}`. Otherwise the result `FAIL` is returned `{5a,b}`.

The second rule `{2a,b}` models suspension if some arguments are not sufficiently instantiated, leaving the clause to be retried when the arguments become more defined. The final rule `{3a,b}` detects failure to match, passing `FAIL` to the OR node. The `Clausename_Commit` rules will be extended in the next section where we introduce the implementation of the run-time test for full GHC.

A GHC predicate with n clauses is translated by this scheme into 5n+1 simple Dactl rules: 1 for the top level rule, 3 per clause for head unification, and two for guard evaluation and committment. Many optimisations which usually reduce the number of Dactl rules are possible and some will be illustrated later. Empty guards or bodies are replaced by the value SUCCEED.

A guard is either a single call or a conjunction represented using the AND function which can be used with any arity. A body can also be either a single call or a conjunction. A single call is represented explicitly, while a conjunction of calls is represented as `Body[b1 b2 … bn]` with one `b` for each body call. When fired it rewrites to the same form as a guard:

```
Body[b1 b2 … bn] => #AND[^*b1 ^*b2 … ^*bn];

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

The AND function monitors its children processes; if any of the goals fails it terminates with `FAIL;` if they all succeed it terminates with SUCCEED; if the arguments are a mixture of uncompleted goal computation and SUCCEED for completed goals, then the special form used on the right-hand side of the middle rule indicates that the node should be suspended as it is, to be re-awoken when any goal completes.

The above example belongs to the most general case where the clauses have overlapping patterns as well as guards. For the other three cases (unguarded clauses with overlapping patterns, and clauses with non-overlapping patterns with and without guards) a more direct translation to a Dactl rewrite rule system is possible. An optimised translation including use of some primitive predicates is illustrated by the following example:

```
primes(Max,Ps) :- true | gen(2,Max,Ns),
                         sift(Ns,Ps).
gen(N,Max,Ns) :- N=<Max | Ns=[N|Ns1], N1:=N+1,
                          gen(N1,Max,Ns1).
gen(N,Max,Ns) :- N>Max | Ns=[].
sift([P|Xs],Zs) :- true | Zs=[P|Zs1],
                 filter(P,Xs,Ys), sift(Ys,Zs1).
sift([], Zs) :- true | Zs=[].
filter(P,[X|Xs],Ys) :- X mod P=:=0
                       | filter(P,Xs,Ys).

filter(P,[X|Xs],Ys) :- X mod P=\=0
                  | Ys=[X|Ys1], filter(P,Xs,Ys1).
filter(P,[],Ys) :- true | Ys=[].
```

The equivalent Dactl program is shown below:

```
Primes[max ps] => #AND[^o1 ^o2],
        o1:*Gen[2 max ns:Var], o2:*Sift[ns ps];
```

```
Gen[n max ns] => #Search[^#OR[^o1 ^o2]],
       o1:#Gen1_Commit[^*Lesseq[n max] n max ns],
       o2:#Gen2_Commit[^*Greater[n max] ns];
Gen1_Commit[SUCCEED n max ns] =>
                      *Result[Body[b1 b2 b3]],
                       b1:Unify[ns Cons[n ns1:Var]],
                       b2:Eval1[n1:Var Plus[n 1]],
                       b3:Gen[n1 max ns1]|
Gen1_Commit[FAIL Any Any Any] => *FAIL;
Gen2_Commit[SUCCEED ns] =>
                             *Result[Unify[ns Nil]]|
Gen2_Commit[FAIL Any] => *FAIL;
Sift[Cons[p xs] zs] => #AND[^b1 ^b2 ^b3],
                      b1:*Unify[zs Cons[p zs1:Var]],
                      b2:*Filter[p xs ys:Var],
                      b3:*Sift[ys zs1]|
Sift[Nil zs] => *Unify[zs Nil]|
Sift[p1:Var p2] => #Sift[^p1 p2];
Sift[Any Any] => *FAIL;
Filter[p Cons[x xs] ys]
       => #Search[^#OR[^o1 ^o2]],
          o1:#Filter1_Commit[^guard1 p xs ys]
                      guard1:*Eval2[Mod[x p] 0],
          o2:#Filter2_Commit[^guard2 p x xs ys],
                      guard2:*Not_eval2[Mod[x p] 0]|
Filter[p Nil ys] => *Unify[ys Nil]|
Filter[p1 p2:Var p3] => #Filter[p1 ^p2 p3];
Filter[Any Any Any] => *FAIL;
Filter1_Commit[SUCCEED p xs ys] =>
                           *Result[Filter[p xs ys]]|
Filter1_Commit[FAIL Any Any Any] => *FAIL;
Filter2_Commit[SUCCEED p x xs ys] =>
      *Result[Body[b1 b2]],
                       b1:Unify[ys Cons[x ys1:Var]],
                       b2:Filter[p xs ys1]|
Filter2_Commit[FAIL Any Any Any Any] => *FAIL;
```

`Primes` needs no head unification, nor has it a guard, so we execute the body directly. `Gen` also needs no head unification so we go directly to guard evaluation. `Sift` has non-overlapping patterns so no `Search` is required. `Filter` has identical patterns, so the code for head unification is shared. Many more optimisations are possible (some of them particular to Dactl) and these are described in Glauert and Papadopoulos (1988).

The unification primitive may be implemented as follows:

```
Unify[x x] => *SUCCEED;
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 appropriare rules for decomposing structures and comparing ground terms. The first rule is used when a term attempts to unify with itself. The next one unifies two variables using the non-root overwriting facility of Dactl to perform the assignment of one variable to the other. We do not use activation markings in the redirection so any nodes suspended on these variables will not be awakened just to suspend again. The last two rules assign a variable to a non-variable term. Here, the use of the activation marking will awake any nodes waiting for the result of this unification. This definition of unification suffices for the case of safe GHC. In the next section we will extend it to perform the required run-time test for general GHC programs.

For completeness we describe finally the implementation of the `otherwise` primitive (see the appendix for a more efficient implementation); its definition in Dactl is the following:

```
Otherwise[Any SUCCEED] => *SUCCEED;
Otherwise[Any Result[body]] => *body;
Otherwise[otherwise FAIL] => *otherwise;
```

An `Otherwise` process is called as 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. `Otherwise` remains suspended until the `OR` processes have reported back. If they have all failed, `Otherwise` fires the clause that uses the `otherwise` primitive; otherwise it fires the body of the clause that committed successfully.

4.2 OR-parallelism

We now show how GHC's full OR-parallelism can be easily and efficiently implemented using Dactl's pointer equality facility. We recall that if user-defined calls are allowed in the guards there is a need for a run-time check to determine whether a variable attempting unification is allowed to do so. If unification could only proceed by binding a non-local variable, unification suspends. This suggests the need for a mechanism to determine at run-time the current environment of the variable as well as the environment where the binding is attempted. If the two coincide, unification is allowed to proceed; otherwise unification suspends. A variable is now represented as `Var[env]` where `env` denotes the current environment of the variable. Variables introduced in a guard are created in a new local environment. When a guard commits, this environment is merged with the calling environment, promoting the new variables to the status of variables in the calling environment.

GHC allows some computation in the body of a clause to proceeed before successful evaluation of the guard and commitment to the clause. Although our model can support this extreme form of speculative evaluation, we do not believe it will be beneficial in general and consider first the case where the body of a clause is only executed after commitment to that clause. As a consequence, new variables introduced in the body are created in the environment of the caller. To implement this scheme, a GHC call is represented as a Dactl function with an additional argument identifying the environment of the call. For example, the GHC query

```
:-p(X), q(X,Y).
```

is represented in Dactl as

```
Initial =>.#AND[^b1 ^b2],
b1:*P[env:E x:Var[env]], b2:*Q[env x y:Var[env]]
```

where `P`, `Q`, `x`, `y` all share the same environment (`Initial` is a Dactl reserved word indicating the initial graph to be rewritten). The clause

```
p(X) :- r(X,Y) | s(Y,Z).
```

is represented as

```
P[inh x] => #P_Commit[inh loc:E ^guard y],
            guard:*R[loc x y:Var[loc]];
P_Commit[inh loc SUCCEED y] =>
                 *S[inh y z:Var[inh]], loc:=inh|
P_Commit[Any Any FAIL Any] => *FAIL;
```

Any new variables appearing in the guard (eg. `y` above) are given the local environment of that guard (`loc`); however, since the corresponding body will be evaluated only after commitment, any new variables in it (eg. `z`) are given the inherited environment of the caller (`inh`). Upon successful evaluation of the guard, `P_Commit` performs the redirection `loc:=inh` thus merging the environment of its guard with the inherited environment of the head; any local guard variables that remained unistantiated will be given the environment of the caller. The decision on whether to instantiate a variable or suspend is taken by the unification primitive which is now represented as `Unify[env t1 t2]` where the `t1` and `t2` are terms and `env` is the environment where unification is attempted. The definition of `Unify` is:

```
{1}   Unify[Any x x] => *SUCCEED;
{2}   Unify[env v1:Var[env] v2:Var[Any]] =>
                         *SUCCEED, v1:=v2|
{3}   Unify[env v1:Var[Any] v2:Var[env]] =>
                         *SUCCEED, v2:=v1;
{4}   Unify[env v1:Var[Any] v2:Var[Any]] =>
                         ##Unify[env ^v1 ^v2]|
{5}   Unify[env v:Var[env] t:(Any-Var[Any])] =>
                         *SUCCEED, v:=*t|
{6}   Unify[env t:(Any-Var[Any]) v:Var[env]] =>
                         *SUCCEED, v:=*t;
{7}   Unify[env t:(Any-Var[Any]) v:Var[env']] =>
                         #Unify[env t ^v]|
{8}   Unify[env v:Var[env'] t:(Any-Var[Any])] =>
                         #Unify[env ^v t];
```

Rules `{2}` to `{4}` cover the case of unifying two variables: if the environment of the `Unify` call is the same with the current environment of any of these variables, that variable is unified with the other one, otherwise unification suspends. Note the optimisation in the fourth rule: although `Unify` suspends on `v1` and `v2` there is no need for it to suspend also on the environments of the variables. Since a body is executed after commitment, the environment of a variable in a guard can change only at that time; however, a guard will commit only when all its local computation has completed successfully. In other words once an attempted unification suspends, it can only resume if the associated variable is instantiated to a non-variable term by some external goal. As a final point in the description of unifying two variables we stress the need for suspending when the environments of the variables and `Unify` are incompatible even if such a unification does not create local bindings. This is because variables could be instantiated too early. Consider the following program:

```
p(X,Y) :- q(X,Y) | true.
p(X,Y) :- r(X) | true.
q(X,Y) :- true | X=Y.
r(X) :- true | X=a.
```

If the unification `X=Y` required by the guard of the first rule for `p` is always allowed to proceed, then the goal

```
:- X=a, p(X,b).
```

always succeeds whereas the goal

```
:- X=a, Y=b, p(X,Y).
```

may fail if the guard `q(X,Y)` executes first. Our implementation, however, respects the concept of *anti-substitution* (Ueda 1986) and treats the two goals the same way. The last four rules define the case where a variable attempts to unify with a non-variable term. Again if the environments of the variable and `Unify` are compatible, unification is allowed, otherwise it suspends. So in the following program

```
:- p(f(Z)), …
p(X) :- q(X) | … .
q(Y) :- true | Y=f(a).
```

the corresponding Dactl code

```
Initial  => #AND[^p …],
            p:*P[env:E Tup["F" z:Var[env]]];
P[inh x] => #P_Commit[inh loc:E ^guard body],
            guard:*Q[loc x],  body: … ;
Q[inh y] => *Unify[inh y Tup["F" A]];
```

will eventually suspend on the call `Unify[e z:Var[e'] A]`, where `e≠e'`, until some other process instantiates `z`. Incidentally note that in some implementations (Levy 1986) the above program would create cross-environment references with certain associated problems.

We will illustrate the execution of a program using a graphical representation to clarify further some of the issues discussed above. The following program

```
:- p(X), q(X,Y).
p(V) :- q(V,U) | r(U,W).
q(X,Y) :- true | X=1, Y=2.
r(X,Y) :- true | X=2, Y=3.
```

has Dactl code given by

```
Initial => #AND[^b1 ^b2],
b1:*P[env:E x:Var[env]],b2:*Q[env x y:Var[env]];
P[env v] => #P_Commit[env loc:E ^guard u],
            guard:*Q[loc v u:Var[loc]];
P_Commit[env loc SUCCEED u] =>
                  *R[env u w:Var[env]], loc:=env|
P_Commit[Any Any FAIL Any] => *FAIL;
Q[env x y] => #AND[^b1 ^b2],
         b1:*Unify[env x 1], b2:*Unify[env y 2]];
R[env x y] => #AND[^b1 ^b2],
         b1:*Unify[env x 2], b2:*Unify[env y 3]];
```

After executing the first rule, the state of the derived graph is shown in fig. 1:

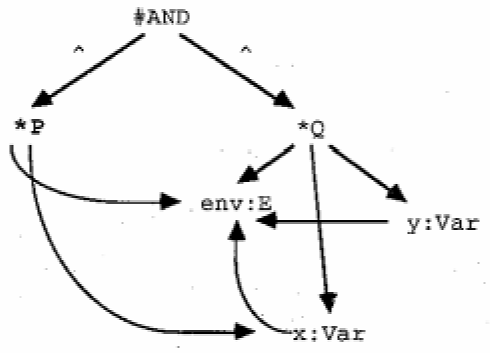

Figure 1

Note that both goals in the conjunction and their associated variables share the same environment. The nodes representing the two goals are both active and can be executed in parallel. Suppose that the one for `P` is executed first. After rewriting it using the appropriate rule, the state of the graph is as shown in fig. 2.

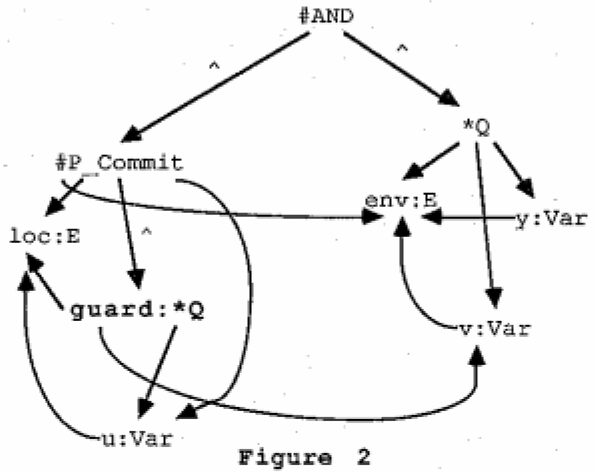

Figure 2

Two nodes are again active, both representing different calls of the predicate, Q. The one shown in bold was introduced in the guard of P and the other is the original call in the goal conjunction. Note here that the former call and all the variables first introduced in it share a different (local) environment. Suppose that the node representing the call in the guard is rewritten first. The graph now looks as shown in fig. 3:

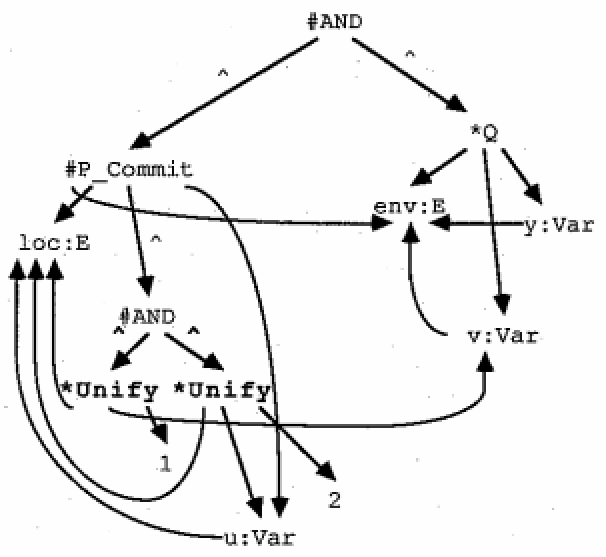

Figure 3

Two unifications are attempted in parallel but only one (u=2) can proceed; the other will have to be suspended until the variable involved, v, is instantiated by the process executing the remaining Q. The left one matches rule 8 in the definition of Unify (given above) and the right one matches rule 5 (fig. 4).

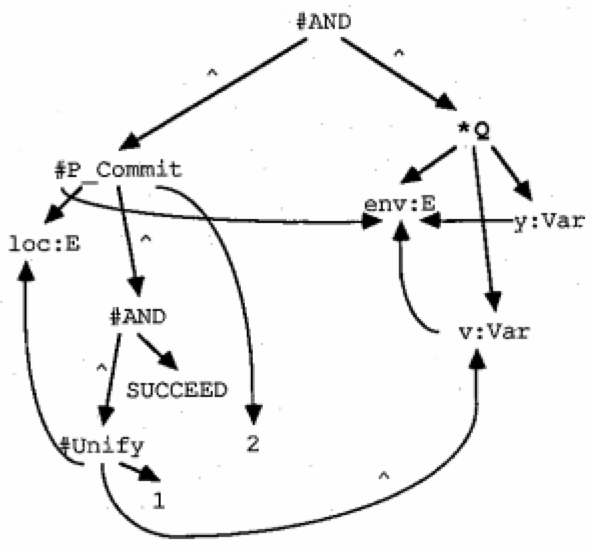

Figure 4

The node corresponding to the unification of the variable u rewrites to SUCCEED and the node for u itself overwrites to 2. However, the second unification has suspended awaiting instantiation of the variable v because the environment of Unify is not the same as the environment of v. The required run-time tests have been performed as simple pointer equality tests. In fact, the only active node is the one corresponding to the goal Q which, when executed, will be able to instantiate the variable v, since Q and v share the same environment. This instantiation will re-activate the node corresponding to the suspended Unify operation which will succeed after testing the compatibility of its two arguments, since v will then be data.

Note that the program used in the example does not conform exactly to the translation scheme outlined earlier. We have introduced the optimisation that for a predicate which has a single guarded clause, we can evaluate the body directly if the guard succeeds. The predicate p is an example of such a predicate.

This optimisation is one example of many which are possible within the framework of Dactl. The appendix to this paper illustrates the code actually generated by our compiler. We explain there some further optimisations.

4.3 Speculative Evaluation

We now extend the above technique so that the body of a clause is executed in parallel with the respective guard which is allowed provided it does not attempt to instantiate a variable either in the guard or the caller of the clause until commitment (second rule of suspension). This is achieved by associating a new local environment with variables introduced in the body. The pointer to the environment of the body is passed to the Commit rules which will promote body variables to the environment of the caller if the clause is chosen. In addition, if we have information that a body fails, we avoid committing to the corresponding clause since it is doomed to fail even if the guard succeeds. Hence we retain GHC sematics, but reduce the failure set somewhat. As an example, let us reconsider the following clause:

```
p(X) :- r(X,Y) | s(Y,Z).
```

This is now represented as follows:

```
P[inh x] => #Commit[inh locg:E locb:E ^g ^b],
            g:*R[locg x y:V[locg]],
            b:*S[locb y z:Z[locb]];
```

Both guard and body are now fired in parallel; unification will again suspend if any incompatibilities in the environments are detected. There is a standard function Commit now which is defined by:

```
(1)  Commit[Any Any Any FAIL Any] => *FAIL|
(2)  Commit[Any Any Any Any FAIL] => *FAIL|
(3)  Commit[inh locg locb SUCCEED SUCCEED] =>
                                    *SUCCEED;
(4)  Commit[inh locg locb SUCCEED body] =>
                     *Result[inh locg locb body]|
(5)  r:Commit[inh locg locb guard SUCCEED] ->
                                          #r;
```

If either the guard or the body fails, we report failure using one of the first two rules. If both the guard and body succeed the third rule is selected which simply reports success. The fourth rule is selected if the guard has successfully terminated but the body is still executing; the Commit rule passes the necessary data to the monitoring Search process which choses a candidate clause and will merge the environments as required. The final rule is selected in the case where the body has succeeded, but the guard has not completed; `Commit` is suspended again waiting for the guard to complete execution. `Search` is now defined as:

```
Search[p:(SUCCEED+FAIL)] => *p;
Search[Result[inh locg locb body]] => body,
                              locg:=*inh, locb:=*inh;
```

The `Unify` primitive must now be modified to suspend also on the environments of the variables; this is because a suspended unification in a body should now resume *either* when an involved variable is instantiated to a non-variable term *or* when, after commitment to the clause, the environment is promoted. Rules {4}, {7} and {8} are modified accordingly and there is an additional auxiliary rule:

```
{4'} Unify[env v1:Var[env1] v2:Var[env2]] =>
          #Unify_wait[env ^v1 ^env1 ^v2 ^env2]|
{7'} Unify[env v:Var[env'] t] =>
          #Unify_wait[env ^v ^env' t env']|
{8'} Unify[env t v:Var[env']] =>
          #Unify_wait[env t env' ^v ^env'];
{9}  Unify_wait[env t1 Any t2 Any] =>
          *Unify[env t1 t2];
```

5 PERFORMANCE

Initial results from analysing the performance of our compiler from GHC to Dactl show that inclusion of the run-time test does not significantly impair performance for safe programs. The programs were run on a Sun 3/180 using our Dactl interpreter written in C which executes approximately 1000 rewrites per second.

In the table below, the first figure in each entry shows the performance of our GHC compiler; the second (in italics) shows the performance of the handwritten code for a PARLOG to Dactl implementation.

Programs	**A**	**R**	**PC**	**GNC**
Merge (100x100)	2021 *2021*	1211 *1211*	614 *614*	2625 *2624*
Primes (1 to 300)	86494 *82013*	58207 *53726*	4558 *4495*	83947 *83947*
Quicksort (50 els reversed.)	27234 *27234*	16879 *16879*	944 *944*	26314 *26313*
Tree search (15 els bin. tree)	200 *216*	143 *148*	65 *61*	368 *349*
Isotree (15 els bin. tree)	893 *812*	649 *568*	193 *177*	1211 *968*

A=Activations, **R**=Rewrites, **GNC**=Graph nodes created, **PC**=Parallel cycles performed.

6 DISCUSSION AND RELATED WORK

The implementation of the flat subset of GHC is very similar to the one for PARLOG which is described in Glauert *et al.* (1988) and Papadopoulos (1988). Since in PARLOG the safety test is done at compile time, there is no need for a run-time test. However, the compile-time test is not always successful; safe programs may be rejected as unsafe. A combination of the compile and run-time tests is possible in our Dactl implementations and it could be useful for both PARLOG and GHC.

Although our implementation detects failure as early as possible it cannot abort a computation whose result is not needed any more; this is because the notion of killing active processes has no meaning in a graph rewriting model. If an AND process detects a failure from one of its children it terminates immediately, reporting failure, but it does not kill the rest of its child processes. In most cases these will eventually suspend waiting for input data which will never arrive. Also, when an OR process commits to the body of a clause, the rest of the guards continue their execution although they cannot affect the final result. This is perfectly acceptable by the definition of the language; as Ueda (1986) points out a truly parallel language may have to allow possible useless computation. However, for efficiency reasons we would like to be able to terminate such unnecessary computation. This is possible in our model by means of a transformation technique which involves enhancing a Dactl function with an additional parameter to be instantiated when the computation of that function must be aborted; indeed, it can also be used to suspend and resume computation, which is useful for systems programming and metaprogramming. This technique is discussed in the implementation of PARLOG (Papadopoulos 1988) and the reader is referred to that document for further details.

Finally, we note that our technique for detecting the current environment of a variable (needed when calls to user defined predicates are allowed in the guards) involves just a simple pointer equality test. We believe that this technique is more efficient than the ones proposed so far such as pointer colouring or guard system numbers (Kishi *et al.* 1985). Commitment is performed by a simple non-root overwrite which merges the local and the calling environment and does not involve the difficulties of other known schemes such as handling cross-environment references (Levy 1986). We would like to stress here the fact that, if desired, a mixture of the two possible strategies (speculative and non-speculative) can be used. Some of the clauses of a GHC program may be translated to Dactl code based the first strategy, and others to code based on the second. Although speculative evaluation of the body of a clause in parallel with the respective guard achieves the maximum degree of parallelism, it remains to be seen whether it is an acceptable strategy from the point of view of efficiency.

7 CONCLUSIONS AND FUTURE RESEARCH

We have presented a parallel implementation of Guarded Horn Clauses based on graph rewriting. We showed that GHC clauses can be seen as rewrite rules specifying possible transformations on graphs representing goal queries. We provided a complete translation scheme from GHC to Dactl, a compiler target language based on graph reduction. This scheme is general enough to accomodate all kinds of GHC programs including the ones that use nested calls in the guards (the required Dactl code to run GHC programs as well as a number of example programs can be found in Glauert and Papadopoulos (1988)). The ability to share subexpressions, an inherent property of graph reduction, allowed the run-time test needed for such programs to be implemented using a simple pointer equality test. An additional advantage of such an implementation stems from the fact that Dactl will be used as a front-end to a number of machines namely Flagship (Watson *et al.* 1988), GRIP (Peyton Jones *et al.* 1987) and ZAPP (Sleep and Kennaway 1984).

The use of Dactl as the implementation language for GHC allows the former to be used as a common basis for comparing implementations of the latter with other languages both logic and functional. This is discussed further by Hammond and Papadopoulos (1988). A performance comparison between our GHC (or PARLOG) to Dactl implementation on Flagship and a similar one for PARLOG on Alice CTL (Lam and Gregory 1987) would also be interesting.

We are currently investigating the extension of our model to accomodate a wider family of languages, namely the **CP**-family, P-Prolog (Yank and Aiso 1986) and a class of equational (logic+functional) languages (DeGroot and Lindstrom 1986).

ACKNOWLEDGEMENTS

We are grateful to the anonymous referees for their constructive comments.

REFERENCES

Barendregt H. P., van Eekelen M. C. J. D., Glauert J. R. W., Kennaway J. R., Plasmeijer M. J. and **Sleep M. R.**, *Term Graph Rewriting*, PARLE Conference, Eidhoven, Netherlands, June 15-19, LNCS 259, pp. 141-158, 1987.

DeGroot D. and **Lindstrom G.** (eds), *Logic Programming: Functions, Relations and Equations*, Prentice-Hall, 1986.

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

Glauert J. R. W., Kennaway J. R. and **Sleep M. R.**, *Dactl: a Computational Model and Compiler Target Language Based on Graph Reduction*, ICL Technical Journal, May, pp. 509-537, 1987.

Glauert J. R. W. and **Papadopoulos G. A.**, *A Parallel Implementation of GHC Based on Graph Reduction*, Internal Report, University of East Anglia, U.K., 1988.

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

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

Kennaway J. R., *Implementing Term Rewrite Languages in Dactl*, CAAP '88, Nancy, France, March 21-24, LNCS 299, pp. 102-116, 1988.

Kishi M., Kuno E., Rokusawa K. and **Ito N.**, *The Dataflow-based Parallel Inference Machine to Support Two Basic Languages in KL1*, ICOT, Japan, TR-114, 1985.

Lam M. and **Gregory S.**, *PARLOG and ALICE: A Marriage of Convenience*, 4th ICLP, Melbourne, Australia, May 25-29, pp. 294-310, 1987.

Levy J., *A GHC Abstract Machine and Instruction Set*, 3rd International Conference on Logic Programming, London U.K., July 14-18, LNCS 225, pp. 157-171, 1986.

Papadopoulos G. A., *A High-Level Parallel Implementation of PARLOG*, Internal Report SYS-C88-05, University of East Anglia, U.K., 1988.

Peyton Jones S. L., Clack C., Salkild J. and **Hardie M.**, *GRIP - A High-Performance Architecture for Parallel Graph Reduction*, FPLCA'87, Portland Oregon, U.S.A., Sept. 14-17, LNCS 274, pp. 98-112, 1987.

Shapiro E. Y., *A Subset of Concurrent Prolog and its Interpreter*, ICOT, Japan, TR-003, 1983.

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

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

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, pp. 124-130, Honolulu, Hawaii, May 30 - June 2, 1988.

Yank R. and **Aiso H.**, *P-Prolog: A Parallel Logic Language Based on Exclusive Relations*, 3rd International Conference on Logic Programming, London U.K., July 14-18, LNCS 225, pp. 255-269, 1986.

APPENDIX

GHC Code

```
search(Key,Value,t(_,l(Key,Value'),_))
                              :- true | Value=Value'.
search(Key,Value,t(X,_,Y)) :- otherwise |
                              search1(Key,Value,X,Y).
search1(Key,Value,X,Y) :- search(Key,Value',X) |
                          Value=Value'.
search1(Key,Value,X,Y) :- search(Key,Value',Y) |
                          Value=Value'.
```

Dactl Code

```
MODULE Search;
IMPORTS Arithmetic; Logic; Lists; Strings; GHC;
SYMBOL REWRITABLE Search; Search_Commit;
                  Search1; Search1_Commit;
PUBLIC Search;
RULE
Search[inh vo v1 Tup["T" v2 Tup["L" v3 v4] v5]]
   => #Search_Commit[inh ^guard v0 v1 v2 v4 v5],
      guard:*Eq[v0 v3]|
Search[p1 p2 p3 p4:Var[Any] p5]
                     => #Search[p1 p2 p3 ^p4 p5];
Search[p1 p2 p3 Tup["T" v2 ^1:Var[Any] v5] p5]
    => #Search[p1 p2 p3 ^#Tup["T" v2 ^1 v5] p5];
Search[Any Any Any Any Any] => *FAIL;
Search_Commit[inh SUCCEED Any v1 Any v4 Any]
                         => #Unify[inh v1 v4]|
Search_Commit[inh FAIL v0 v1 v2 Any v5]
                  => *Search1[inh v0 v1 v2 v5];
Search1[inh v0 v1 v2 v3]
     => #Search1_Commit[inh loc1:E loc2:E
              ^*Search1[loc1 v0 v4:Var[loc1] v2]
              ^*Search1[loc2 v0 v5:Var[loc2] v3]
         v1 v4 v5];
Search1_Commit[inh loc1:E Any SUCCEED Any
              v1 v4 v5] => *Unify[inh v1 v4],
                           loc1:=inh|
Search1_Commit[inh Any loc2:E Any SUCCEED
              v1 v4 v5] => *Unify[inh v1 v5],
                           loc1:=inh|
Search1_Commit[Any Any Any FAIL FAIL
              Any Any Any => *FAIL;
r:Search1_Commit[Any Any Any Any Any Any Any
                                   Any] -> #r;
ENDMODULE Search;
```

Note how the functionality of `otherwise` is embedded implicitly in the rules defining `Search` instead of using the explicit, more inefficient, representation described in the paper. Note also the optimisation in `Search_Commit` where no local environment is created for `Eq` because it is a safe system primitive. This optimisation can be extended to guards having user defined calls that are known to be safe (possibly by means of a compile time test).