UNIFYING CONCURRENT LOGIC AND FUNCTIONAL LANGUAGES IN A GRAPH REWRITING FRAMEWORK

John R. W. Glauert and George A. Papadopoulos
Declarative Systems Project, School of Information Systems,
University of East Anglia, Norwich NR4 7TJ, U.K.
e-mail: {jrwg/gp}@sys.uea.ac.uk

ABSTRACT

We examine the extension of a concurrent logic language with some capabilities found normally only in functional languages using the intermediate language approach. In particular, we show how the language Guarded Horn Clauses (GHC) can be extended with eager and lazy functions, sharing of computations and higher-order function/predicate applications. This is achieved by superimposing a functional sub-language having these capabilities on the top of GHC. Both components, the concurrent logic one and the functional one are compiled down to Dactl, a compiler target language based on generalised graph rewriting. Thus, the underlying representation enjoys a uniformity which is useful both at the integration and the implementation level.

KEYWORDS

Logic Programming, Functional Programming, Amalgamation of Logic and Functional Languages, Graph Reduction (Rewriting), Parallel Processing.

1. INTRODUCTION

There have been many attempts to amalgamate logic and functional languages ([7]). There are basically 4 approaches: interface existing languages ([18]), extend functional languages with logic capabilities ([19]), extend logic languages with functional capabilities ([16]), and finally, design new languages based on some common computational model such as equational theories ([2,20]), or various forms of narrowing ([8]). Our approach lies somewhere between the last two. On the one hand, we extend the concurrent logic language GHC ([21]) with lazy evaluation and sharing of computation by superimposing a suitable parallel functional sub-language on the top of it. On the other hand, however, both the concurrent logic and functional components are mapped to sets of rewrite rules written in Dactl, a compiler target language based on generalised graph rewriting. Thus, both components enjoy an underlying uniformity at both the computational model level (i.e. graph reduction) and the architectural level (suitable reduction machines).

This paper attempts to contribute to current research in this area by:

— illustrating the usefulness of some characteristic features of functional languages such as lazy evaluation and sharing in a concurrent logic programming framework;
— providing an interaction between the "eager" predicates of a concurrent logic language with the lazy or eager functions of its functional component which offers the programmer a variety of execution strategies and explicit control over space-time trade-offs;
— showing how to execute both functions and predicates using graph reduction as a common underlying execution mechanism, thus enjoying uniformity in data structures (graph nodes representing data terms), value passing mechanism (matching of redexes and arc redirection modelling variable instantiation), control mechanism (graph rewriting), and architectural support (reduction machines);
— examining the potential of graph reduction to act as a bridge in unifying logic and functional languages.

The rest of the paper is organised as follows: the next two sections introduce the reader to the enhanced version of GHC (henceforth referred to as GHC/F) and Dactl. The following two present some examples showing the expressiveness of our model and describe their implementation in Dactl. The final one provides some conclusions and related research.

2. GHC/F

A GHC program is a set of (possibly guarded) clauses of the form:

```
p(t1,t2,…,ti) :- g1(…),…,gm(…) | b1(…),…,bn(…).
```

where `g1,…,gm` and `b1,…,bn` are calls to predicates or to the unification primitive '='. A goal of the form `p(s1,s2,…,si)` attempts to reduce with all its defining clauses by unifying with the heads of these clauses and solving of any guards. If during unification, an attempt is made to instantiate a variable in the call with a non-variable term in the head, unification suspends until some other process running in parallel with the goal in question instantiates the variable. Eventually the goal will reduce to the set of goals in the body of that clause that manages to complete successfully head unification and guard evaluation (if there are more than one candidates, a single clause will be selected non-deterministically). Note that the guards must satisfy a *safety* property that requires them to refrain from instantiating any variables in the environment of the caller until commitment to their respective bodies. This safety test is done in GHC at run-time and involves checking the compatibility of the environment where unification is attempted with the environments of the variables involved in the unification. The way the run-time test is supported in our model is described elsewhere ([11]).

Programs in GHC/F consist of clauses of the form shown above, plus (possibly conditional) rewrite rules of the following form:

```
        LHS => RHS          or        LHS => RHS :- G | B
```

where `LHS` and `RHS` are function applications and the part '`:- G | B`' is called the *condition* where `G` and `B` are defined as in GHC. GHC/F supports 3 unification primitives:

'=' denotes lazy evaluation; `x=f(…)` will unify `x` with `f(…)` without evaluating `f` (this allows sharing of computations as we will see below);

':=' is the eager assignment primitive; `x:=f(…)` will assign `x` to `f(…)` and at the same time fire `f`; if at the time of the call `x` is not a variable, an error will be reported;

'=:=' is the eager unification primitive; `s(…)=:=t(…)` will attempt to unify `s(…)` with `t(…)` evaluating them if needed (if either or both represent function applications instead of data terms).

Functions can appear anywhere in the program where data terms are allowed with one exception: they are not allowed in the head of a clause or as arguments to the functions comprising the `LHS` of '`=>`' (free constructor discipline). Evaluation of functions is done in a combined eager and lazy manner, in the following sense: A function application attempts to reduce using its defining rewrite rules. The defining rules are all tried in parallel, and for those of them that have a condition part '`:- G | B`', `G` is also evaluated in parallel. In addition, head matching is also done in parallel. However, function evaluation is done lazily in a *pattern-driven* way ([20]). If an attempt is made to match an evaluable argument of a function with a corresponding non-variable argument in the lhs of a defining rewrite rule, the inner function call is evaluated enough for matching to proceed. Note that if an attempt is made to instantiate a variable in the function call with a non-variable term in the lhs of the rewrite rule, matching suspends. Eventually, exactly one rule should be candidate for reducing the function call (that one for which matching and evaluation of any guard condition terminates successfully); the call is then reduced to the rhs of the rule, and if the rule is conditional with a body part `B`, evaluation of `B` starts in parallel – this must always succeed. Although evaluation of functions is lazy by default, it is possible to indicate that the values of certain evaluable arguments or function applications are needed (and therefore they could be evaluated eagerly) using the two strict unification primitives discussed above. This is similar to strict annotations used in many lazy functional languages ([4,5,13]). We illustrate the use of the above concepts after introducing Dactl.

3. DACTL

The language Dactl provides a notation for describing computational objects in terms of directed graphs and for describing programs in terms of pattern-directed graph transformation rules. Dactl may be used as a vehicle for comparing implementation techniques and computational strategies and also, since it has a well-defined operational semntics, as an intermediate code for language implementation. Studies at UEA have led to implementations of GHC and the functional languages Clean ([15]) and Standard ML. Within the Alvey Flagship project Dactl has been used to implement Parlog ([17]) and Hope+.

The nodes of a Dactl graph are labelled with a *symbol* which indicates that the node plays the role of an *operator* at the root of a rule application, a data *constructor*, or an *overwritable*. The use of overwritable nodes which may be modified as a side-effect of a rule application enables the generalised graph rewriting of Dactl to express many more computations than the conventional graph rewriting used to implement functional languages. Overwritables may model von Neumann storage cells, semaphores, and the logic variable (sufficient for concurrent logic languages at least). Each node also has a distinct identifier. From a node leads a sequence of arcs to *successor* nodes. Arcs point from an operator node to its arguments, or from a data constructor or overwritable to its sub-fields.

A Dactl graph may be represented by listing the definitions of the nodes giving their identifier, symbol, and a sequence of identifiers for the successor nodes. Repetition of identifiers is used to indicate sharing in a graph:

```
a: Append[ l l ],
l: Cons[ o n ],
o: 1,
n: Nil
```

Symbols are integers or identifiers starting in upper-case, while node identifiers start in lower-case. A node definition may replace one of the occurrences of the node identifier, and redundant identifiers may be removed allowing the shorthand form:

```
a: Append[ l:Cons[ 1 Nil ] l ]
```

Dactl rules contain a *pattern* to be matched and a body, or *contractum*, to replace the occurrence of the pattern in the graph, or *redex*. Patterns are Dactl graphs but may contain *pattern operators* in addition. A pattern which is a standard graph will match a subgraph with the same form, or one with more sharing in some cases. The pattern operator ANY may appear in place of a node and will match an arbitrary node. The operators '+', '-', and '&', may be used to combine patterns: the pattern 'P1+P2' matches if either P1 or P2 match; 'P1-P2' matches if P1 matches but not P2; and 'P1&P2' matches if both P1 and P2 match.

The contractum of a rule contains new graph structure to be built, which may reference nodes matched by the pattern, and one or more *redirections*, which indicate (roughly) that the source of the redirection should be overwritten by the target.

Computation under the Dactl model proceeds by identifying a subgraph matching the pattern of a rewriting rule and replacing it by the contractum of the rule. If more than one rule matches, an arbitrary choice may be made about which rule to apply; fairness is not assumed. To control the order of evaluation, attempts to match the rules against the graph only begin at *active* nodes. Such nodes are marked with a '*' in the representation of a Dactl graph. The pattern of a rule contains no such markings, since the matching process is insensitive to markings, but the contractum may use markings to nominate further nodes at which rewriting may take place. If multiple active nodes arise they may be considered in any order, or even in parallel if there is no conflict between possible rewritings.

Nodes may also be created *suspended* waiting for *notification* that a successor has been rewritten to a stable form. This enables a rule to create a dataflow graph in which certain nodes are active and will produce results which awaken parent nodes once all arguments are available. Suspension is indicated by one or more '#' markings. Each notification removes one '#', the node becoming active when the last '#' is removed. Notification takes place when an active node is considered for rewriting, but not rule matches. This is typically because the node in question is a constructor or overwritable. This explains the counter-intuitive form of some Dactl rules in which a node is overwritten by an active constructor; when matching is attempted, no rule will match, the '*' marking is removed, and notifications will travel up to any suspended parent node. Arcs which will form notification paths are marked with '^'.

The following examples show some rules with markings and the execution sequence for functional versions of append, and a version using the logic variable.

```
FAppend[Cons[u x] y] => #Cons[u ^*FAppend[x y]]        |       {1}
FAppend[Nil y] => *y                                   ;       {2}
FAppend[p1 p2] => #FAppend[^*p1 p2]                    ;       {3}
```

Identifiers with no definition, such as u, x, and y above, are associated with the pattern operator ANY. The example is a strict function which rewrites to a data constructor node, Cons, which is suspended waiting for completion of a recursive invocation of the function before notifying the parents of the original node. The final rule applies if the first argument has not been rewritten to the form of a list. The argument is activated and the FAppend application will be reconsidered when the argument has been rewritten. The separators between rules are significant. The bar indicates that the first two rules may be tested in parallel, but the semicolon indicates that neither of these may apply before testing for the third rule. The semicolon has an effect exactly like the use of the pattern operator '-' to exclude the patterns of the first two rules.

The first rule could be modified in a number of ways to achieve different behaviour:

```
FAppend[Cons[u x] y] => *Cons[u FAppend[x y]]                  {1'}
FAppend[Cons[u x] y] => *Cons[u *FAppend[x y]]                 {1"}
```

Rule {1'} is a lazy version since the data constructor is made active and will therefore notify parent nodes of the original active node immediately. Rule {1"} is an eager version; a result is returned immediately, as in the case of lazy evaluation, but evaluation of the tail of the list proceeds concurrently.

We will illustrate evaluation of a Dactl graph using an example with sharing, indicating the rule applied at each step:

```
*FAppend[ l:Cons[ o:1 n:Nil ] l ]                         => {1}
#Cons[ o:1 ^s:*FAppend[n l:Cons[o n:Nil]] ]               => {2}
#Cons[ o:1 ^l:*Cons[o n:Nil] ]                            => {NoMatch}
*Cons[ o:1 l:Cons[o n:Nil] ]                              => {NoMatch}
Cons[ o:1 l:Cons[o n:Nil] ]
```

The computation terminates when no active nodes remain.

The following rules illustrate a version of append from logic programming. Here the result of the rewrite indicates success or failure of a predicate and results are communicated by instantiation of shared variables.

```
LAppend[Cons[u x] y r:Var] => *LAppend[x y w:Var], r:=*Cons[u w]        |     {1}
LAppend[Nil y r:Var] => *SUCCEED, r:=*y                                 |     {2}
LAppend[p:Var q r:Var] => #LAppend[^p q r]                              ;     {3}
```

The result of the first rule depends on the success of a recursive use of the predicate. The variable r is bound to a Cons node whose second argument is a new variable which will be instantiated by the recursive call. The Cons node is made active in order to notify any computation suspended waiting for the variable to be instatiated. The final rule illustrates the case when the critical first argument is not instantiated. The call suspends waiting for some other computation to instantiate the variable (using one of the first two rules, for example).

We illustrate these rules in action with two calls to LAppend which share a variable:

```
*LAppend[Nil Cons[1 Nil] v:Var],  *LAppend[v Cons[2 Nil] w:Var]    => {3}
*LAppend[Nil Cons[1 Nil] v:Var],  #LAppend[^v Cons[2 Nil] w:Var]   => {2}
*SUCCEED, v:*Cons[1 Nil],  #LAppend[^v Cons[2 Nil] w:Var]          => {NoMatch}
*SUCCEED,  *LAppend[v:Cons[1 Nil] Cons[2 Nil] w:Var]               => {1}
*SUCCEED,  w:*Cons[1 x], *LAppend[Nil Cons[2 Nil] x:Var]           => {2}
*SUCCEED,  w:*Cons[1 x], x:*Cons[2 Nil], *SUCCEED                  => {…}
```

The third step, when there is no match on the value assigned to variable v, reactivates the second LAppend predicate.

4. PROGRAMMING IN GHC/F

The way a programmer can take advantage of combinations of lazy and eager evaluation, sharing of computation, etc. is best illustrated by means of examples.

We start by reconsidering the append program and its representation at the GHC/F level:

```
append([U|X],Y) => [U|append(X,Y)].
append([],Y) => Y.
```

This program appends two lists lazily; the recursive append in the first rule will be executed only if some head or body unification requires its result (pattern-driven). However, the following one

```
length([]) => 0.
length([_|T]) => 1+L :- true | L:=length(T).
```

is eager. Note the use of ':=' to force the evaluation of the recursive call to length and compare the above program with the one that follows

```
length([]) => 0.
length([_|T]) => 1+length(T).
```

which is lazy.

The next example computes efficiently the Fibonacci numbers using extra variables for storing intermediate results ([14]):

```
fib(0,X,_) => 1 :- true | X:=1.
fib(1,X,Y) => 1 :- true | X:=1, Y:=1.
fib(N,X,Y) => fib(N1,Y,Z) :- true | N1:=N-1, X:=Y+Z.
```

The argument X is used to store the result of the computation. Note that in the last rule, the values for N1 and X are computed in parallel with the recursive call to Fib. In fact, the above program is more efficient than the fully lazy version given in [14].

The following is an implementation of the quicksort algorithm .

```
sort(List,Sorted) :- true | qsort(List,Sorted,[]).

qsort([U|X],Sorted_h,Sorted_t) :- true | X1:=partition(U,X,X2),
                                         qsort(X1,Sorted_h,[U|Sorted]),
                                         qsort(X2,Sorted,Sorted_t).
qsort([],Sorted_h,Sorted_t) :- true | Sorted_h=Sorted_t.

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

Note here the use of the function partition that reduces itself to the first sublist while one of its arguments is instantiated to the second sublist. Note also that qsort is a predicate that concatenates the two sublists in constant time using difference lists. The above program is one of the most efficient parallel versions we have encountered in the literature so far.

In the next example which is a hamming numbers generator we illustrate the use of sharing and the handling of infinite data structures. Consider the first version:

```
hamming() => H :- true | H=[1|merge(times(2,H),merge(times(3,H),times(5,H)))].

merge([U|X],[V|Y]) => [U|merge(X,Y)] :- U==V | true.
merge([U|X],[V|Y]) => [U|merge(U,[V|Y])] :- U<V | true.
merge([U|X],[V|Y]) => [V|merge([U|X],Y)] :- V<U | true.

times(U,[V|Y]) => [U*V|times(U,Y)].
```

The above version computes lazily the infinite list of hamming numbers. Note the use of '=' that creates cycles sharing the parts of the list already computed. Although some parallelism exists in the definition of merge (where head matching and evaluation of guards is done in parallel), the program is unnecessarily sequential: U*V can be evaluated in parallel with the reporting of the cons structure; in addition, merge's arguments are known to be needed before merge is actually evaluated. Consider now the following version:

```
hamming() => H :- true | H=[1|merge(X2,merge(X3,X5))],
                          X2:=times(2,H), X3:=times(3,H), X5:=times(5,H).

merge([U|X],[V|Y]) => [U|merge(X,Y)] :- U==V | true.
merge([U|X],[V|Y]) => [U|merge(X,[V|Y])] :- U<V | true.
merge([U|X],[V|Y]) => [V|merge([U|X],Y)] :- V<U | true.

times(U,[V|Y]) => [W|times(U,Y)] :- true | mul(U,V,W).
```

Here, the three calls to times are performed in parallel with the reporting of the cons structure. In addition, the function U*V has been replaced by the predicate mul(U,V,W) which in addition to computing W immediatelly, it also makes the program more expressive: since predicates use unification, the above program can also be used to verify whether the elements of a given list are hamming numbers. So the goal [1,X,3,Y|Z]=:=hamming() would succeed verifying that 1 and 3 are the first and third hamming numbers, binding the variables X and Y to 2 and 4 respectively and Z to some unevaluated expression.

Finally, we show how higher-order programming is supported in our model. The well-known mapping function can be written both as a function and predicate:

```
f_map(_,[]) => [].
f_map(F,[U|X]) => [f_appl|f_map(F,X)] :- true | f_appl:=F@(U).

r_map(_,[],Y) :- true | Y=[].
r_map(R,[U|X],Y) :- true | Y=[V|Y1], R@(U,V), r_map(R,X,Y1).
```

where '@' is a meta-symbol used for function or predicate application. Note that f_map is only partially lazy since the first element of the list will be computed completely (but only that argument); a completely eager or lazy version can also be written using the techniques shown above. Finally, we provide the definition of the folding function:

```
fold(_,R,[]) => R.
fold(F,A,[U|X]) => fold(F,R,X) :- true | R:=F@(A,U).
```

Note again here that fold is only partially lazy. Thus, the following call

```
:- fold(F,R,f_map(G,ints1(M,N))).
```

where ints1(M,N) produces lazily the list of integers in the range M to N will compute concurrently the function applications F and G modelling, in effect, a buffer of size one.

Finally, consider the following example:

```
square(N) => N*N.

hamsql(N) => f_map(square,Temp_1) :- true | Temp_1:=first_n_ham(N,hamming()).

first_n_ham(0,_) => [].
first_n_ham(N,[H|T]) => [H|T1] :- N≠0 | N1:=N-1, T1:=first_n_ham(N1,T).
```

Note that as the head of the list of hamming numbers is passed to f_map the recursive call to first_n_ham generates the rest of the list in parallel (thus achieving the effects of call-by-opportunity, [4]).

5. MAPPING GHC/F PROGRAMS TO DACTL REWRITE RULES

We show now how the above programs are translated to Dactl. Attention will be paid to the special features of our computational model. The interested reader is invited to consult references [9,11,15,17,22] for details of mapping concurrent logic and functional languages to Dactl. We start with the append program:

```
Append[Cons[u x] y] => *Cons[u Append[x y]] |
Append[Nil y] => *y|
Append[p1:Var p2] => #Append[^p1 p2];
Append[p1 p2] => #Append[^*p1 p2];
```

The third rule is used for suspending the computation if a needed argument has not been instantiated yet. The final, default, rule fires an unevaluated expression. In general, unconditional rules of the form 'LHS => RHS' are translated to Dactl as 'LHS => *RHS'. The eager function length is translated as follows:

```
Length[Nil] => *0|
Length[Cons[ANY t]] => *Add[1 *Length[t]] |
Length[p:Var] => #Length[^p];
Length[p] => #Length[^*p];
```

Note that ':=' has, in fact, been eliminated in the Dactl version (it is implicit in the use of the activation marking '*' in the recursive call to Length in the second rule). In many cases where the three unification primitives act as control markings on function calls and their arguments or for denoting sharing, a more concrete representation of them is possible at the Dactl level.

We now turn our attention to the Fibonacci function:

```
Fib[0 x:Var ANY] => *1, x:=*1|
Fib[1 x:Var y:Var]=> *1, x:=*1, y:=*1|
Fib[n:(Int-0-1) x:Var y] => *Fib[*Sub[n 1] y z:Var], x:=*Add[y z] |
Fib[p1:Var p2 p3] => #Fib[^p1 p2 p3];
Fib[p1 p2 p3] => #Fib[^*p1 p2 p3];
```

Again note here the elimination of ':=' in the rhs of the third rule. Note also the representation of new variables in the rhs of rules as nodes with the pattern Var and Dactl's flexibility in instantiating them to either ordinary values or even evaluable expressions using non-root overwrites.

The same techniques are used in the translation of the quicksort program:

```
Sort[list sorted] => *Qsort[list sorted Nil];

Qsort[Cons[u x] sorted_h sorted_t] => #AND[^o1 ^o2],
                                                o1:*Qsort[*Partition[u x x2] sorted_h Cons[u sorted]],
                                                o2:*Qsort[x2:Var sorted:Var sorted_t] |
Qsort[Nil sorted_h sorted_t] => *Unify[sorted_h sorted_t] |
Qsort[p1:Var p2 p3] => #Qsort[^p1 p2 p3];
Qsort[p1 p2 p3] => #Qsort[^*p1 p2 p3];

Partition[u Cons[v x] x2] => #Partition_Rewrite[^g1 ^g2 u v x x2],
                             g1:*Less[u v], g2:*Lesseq[v u] |
Partition[ANY Nil x2:Var] => *Nil, x2:=*Nil|
Partition[p1 p2:Var p3] => #Partition[p1 ^p2 p3];
Partition[p1 p2 p3] => #Partition[p1 ^*p2 p3];

Partition_Rewrite[SUCCEED ANY u v x x2] => *Cons[v *Partition[u x x2]] |
Partition_Rewrite[ANY SUCCEED u v x x2:Var] => *Partition[u x x2'],
                                               x2:=*Cons[v x2':Var];
r:Partition_Rewrite[ANY ANY ANY ANY ANY ANY] -> #r;
```

Note that the progress of executing a parallel and-conjunction of predicates is monitored by and AND process which reports either when all of them complete their execution successfully, or when any of them fails. Note also that the identical pattern matching in the first two GHC/F rule for `Partition` is executed only once. The last rule for `Partition_Rewrite` suspends if some guards have failed while some others are still executing.

We now turn our attention to the second version of the hamming program:

```
h:Hamming => *Cons[1 Merge[*Times[2 h] Merge[*Times[3 h] *Times[5 h]]]];

Merge[l1:Cons[u x] l2:Cons[v y]] => #Merge_Rewrite[^g1 ^g2 ^g3 u v x y l1 l2],
                                    g1:*Eq[u v], g2:*Less[u v], g3:*Less[v u];
Merge[p1:(Var+Cons[ANY ANY]) p2:(Var+Cons[ANY ANY])] => #Merge[^p1 ^p2];
Merge[p1:REWRITABLE p2:REWRITABLE] => ##Merge[^*p1 ^*p2];
Merge[p1:REWRITABLE p2] => #Merge[^*p1 p2] |
Merge[p1 p2:REWRITABLE] => #Merge[p1 ^*p2];

Merge_Rewrite[SUCCEED ANY ANY u v x y l1 l2] => *Cons[u Merge[x y]] |
Merge_Rewrite[ANY SUCCEED ANY u v x y l1 l2] => *Cons[u Merge[x l2]] |
Merge_Rewrite[ANY ANY SUCCEED u v x y l1 l2] => *Cons[v Merge[l1 y]];
r:Merge_Rewrite[ANY ANY ANY ANY ANY ANY ANY ANY ANY] -> #r;

Times[u Cons[v y]] => *Cons[w Times[u y]], *Mul[u v w:Var] |
Times[p1 p2:Var] => #Times[p1 ^p2];
Times[p1 p2] => #Times[p1 ^*p2];
```

Note here how easily sharing is represented in Dactl. Note also that, in general, conditional rules of the form '`LHS => RHS :- true | b1(…),…,bn(…)`' are translated to Dactl as '`LHS => *RHS, *B1[…],…,*Bn[…]`'; no `AND` processes are needed since the `b`'s are expected to succeed. However, if the condition had a guard part, an `AND` process would be used for an and-conjunction there to verify that the condition is satisfied before commiting to the rhs of the rule.

The translation of the final example is shown below.

```
Square[n:Int] => *Mul[n n] |
Square[v:Var] => #Square[^v];
Square[r] => #Square[^*r];

Hamsql[n] => *F_map[Square *First_n_ham[n Hamming]];

First_n_ham[0 ANY] => *Nil|
First_n_ham[n:(Int-0) Cons[h t]] => *Cons[h *First_n_ham[*Sub[n 1] t]] |
First_n_ham[p1:(Var+Int) p2:(Var+List)] => #First_n_ham[^p1 ^p2] |
First_n_ham[p1:REWRITABLE p2:REWRITABLE] => ##First_n_ham[^*p1 ^*p2];
First_n_ham[p1:REWRITABLE p2] => ##First_n_ham[^*p1 p2];
First_n_ham[p1 p2:REWRITABLE] => ##First_n_ham[p1 ^*p2];
```

We do not show here the implementation of `F_map`, `R_map` and `Fold` which are implemented at a lower level for efficiency reasons.

We conclude this section with the implementation of '=' (`Unify`) and '=:=' (`FUnify`).

```
Unify[x x] => *SUCCEED;
Unify[v:Var vr:(Var+REWRITABLE)] => *SUCCEED, v:=vr|
Unify[vr:(Var+REWRITABLE) v:Var] => *SUCCEED, v:=vr|
Unify[v:Var t:(ANY-Var-REWRITABLE)] => *SUCCEED, v:=*t|
Unify[t:(ANY-Var-REWRITABLE) v:Var] => *SUCCEED, v:=*t|
Unify[Cons[h1 t1] Cons[h2 t2]] => #AND[^*Unify[h1 h2] ^*Unify[t1 t2]];
```

plus additional rules for decomposing other data structures, comparing ground values, etc. `FUnify` is different in that it forces evaluation of functions; rules 2-5 are replaced by the following ones:

```
FUnify[v1:Var v2:Var] => *SUCCEED, v1:=v2|
FUnify[v:Var t:(ANY-Var)] => *SUCCEED, v:=*t|
FUnify[t:(ANY-Var) v:Var] => *SUCCEED, v:=*t|
```

There is one case that the rules for `Unify` do not cover: if, in addition to a call `v=f(…)`, where `f(…)` is a function call, some other process is suspended on `v` waiting for its value, then we should evaluate `f(…)` even if '`=`' is lazy. One solution is for the waiting process to change the pattern of the variable from `Var` to `NVar`; `Unify` then will recognize that the value for `NVar` must be computed and will fire `f(…)`. This complication, however, does not arise very often since most of the uses of calls such as `v=f(…)` are for denoting sharing and generally the more concrete representation `v:F[…]` which eliminates the variable `v` is possible at the Dactl level. For simplicity of presentation we have not considered the above case in the compilation of the above examples to Dactl.

6. PERFORMANCE

We compare three versions of Hamming and Quicksort written in Clean, GHC and GHC/F. The Clean to Dactl programs have been translated using the compiler described in [15] and they are both lazy. The GHC to Dactl programs have been translated using the compiler described in [11]; lazy evaluation is achieved in GHC by using the technique of making the producer, a consumer of a list of unistantiated variables that will be assigned to the result of the computation. Hamming produces the first 30 elements of the infinite list, and Quicksort sorts a reversed list of 50 elements.

Langs\Progs	**Hamming**				**Quicksort**			
	R	**PC**	**AvP**	**MxP**	**R**	**PC**	**AvP**	**MxP**
Clean	672	1018	1.14	4	6654	11357	1.11	2
GHC	3471	1098	4.98	30	16879	944	28.85	60
GHC/F	1202	736	2.60	9	10255	844	19.77	45

R: Rewrites, **PC**: Parallel cycles performed, **AvP**: Activations processed per cycle (mean value), **MxP**: Activations processed per cycle (peak value).

R is the number of rewrites performed and can be used to measure the complexity of the task performed by the Dactl program. **PC**, the number of parallel cycles performed, assumes the existence of an infinite number of processors (an option for specifying the number of processors available does exist however). **AvP** and **MxP**, measure the average and maximum number of active nodes at each parallel cycle, i.e. how many processors could be employed if they were available. It is clear that the GHC/F versions are more efficient than the GHC ones, and at the same time exhibit more parallelism than the Clean versions.

7. CONCLUSIONS AND RELATED RESEARCH

We have presented an extension of a concurrent logic language with some facilities found only in functional languages. In particular, we showed how true lazy evaluation, sharing of computations and higher-order programming can be integrated into a concurrent logic programming framework in a useful way. By "true lazy evaluation" we mean that computation will not be initiated until needed, but once it does, it is expected to be performed (see, however, the note below on speculative parallelism). This contrasts with the usual techniques of simulating demand driven computation in a concurrent logic language by reversing the producer-consumer relationship and using back communication. Although that technique achieves the desired behaviour, it is still eager as far as the life span of processes is concerned (the producer will remain suspended throughout the computation waiting for its input stream to be instantiated to another sublist); in addition, sometimes difficult to estimate boundary conditions must be specified to close the producer-consumer channels and terminate computation. Finally, the program becomes less clear. Our technique also contrasts with the sort of lazy evaluation found in languages like LEAF and LPL ([2,12]). These languages do not support true lazy functions; instead their implementation model supports an *atom elimination* or *termination* rule that refrains from executing processes which do not satisfy certain conditions (their output is not input to some other process, etc.). This requires their implementation model to have a global view of the state of computation at any time, something that may prove difficult in a highly parallel (or, especially, distributed) environment. In our model, however, such a strong coupling between processes is not required.

In a way our work attempts to approach the subject of amalgamating logic and functional languages from the opposite direction described in [1]. There, a normal-order language has been extended to cope with trial unification achieving the effects of committed-choice logic programming. Here we take the other approach by extending a concurrent logic language with functional capabilities. Our experience shares the view expressed in [1] that combination of unification and "eager" predicate computation with lazy functional evaluation is a very powerful computation model which, however, needs care to handle. Note that in our model it is the user's responsibility to make sure that this combination and the use of strict primitives such as ':=' and '=:=' does not lead to infinite computations performed (if that is not desirable) or deadlock. Note also that we allow these strict primitives to be applied to either function calls or any of their arguments (by flattening them out before applying the primitives to them; similar to the use of strictness annotations). However, in our model it is the user's, rather than the compiler's, responsibility to indicate needness. It is also worth pointing out that our model supports the notion of *irrelevant task* ([13]) in the sense that in the following program

```
:- f(X), X:=g(0).

f(_) => 1.
g(y) => g(y1) :- true | y1:=y+1
```

the value 1 will be returned even if the function g carries on executing indefinitely.

Suitable uses of these primitives offer the user a variety of execution strategies namely call-by-need, call-by-value, call-by-name and call-by-opportunity ([4]), whereas call-by-speculation is offered by default via the committed-choice or-parallelism of the concurrent logic component (note that for the latter, it is possible to terminate speculative computation that is known to be useless or irrelevant; this is descibed in [17]). Their proper use allows the programmer a useful control over the space-time trade-offs; he may choose to either pass around a function application f(X) or evaluate it depending on the size of the data structure represented by X and the overhead in evaluating f. In addition, the stream and-parallelism of the concurrent logic component provides a natural producer-consumer synchronisation and offers incremental computation, thus avoiding the need for intermediate data structures. This minimum of three primitives (indeed, '=:=' is not needed since it is a combination of '=' and ':=') offers the user great flexibility without making the programs difficult to be read and understood as it is, we believe, the case for other annotations proposed or the use of special strict and semi-strict functions.

The extension of a concurrent logic language with functional capabilities was also discussed in an earlier definition of PARLOG ([6]). There, the functions were eager by default (although lazyness in the evaluation of a function could be indicated by means of annotations). It is not clear whether the functional part allowed for sharing and cyclic structures and an implementation of it proved difficult and was subsequently dropped. Here we show how by mapping both the logic and the functional part onto the same intermediate level (Dactl) using a single computational model (graph rewriting) we can provide a homogeneous and efficient implementation. Of course, our PARLOG to Dactl implementation could also benefit from a functional extension of the sort desribed in this paper.

Although the enhancement of GHC with functional capabilities was done in a rather *ad hoc* manner we believe we have shown by means of examples how powerful the model can be. Although this has allowed us to form more general rewriting systems than it may have otherwise been possible (our model, for example, subsumes the one in [14]), development of formal semantics for the model is considered as an area of future research. Note also that we do not consider GHC/F as a finished product but as a vehicle for further research in the area of extending concurrent logic languages with functional capabilities.

It is worth pointing out that our research group is currently extending the work done in Dactl so that a number of Dactl-based implementations of parallel logic and functional languages can be supported by the EDS (European Declarative Systems) machine; an enhanced coarser grain form of graph rewriting (Extended Graph Rewriting) will be used for that purpose that will attempt to solve the problems inherent in fine-grain parallelism.

8. REFERENCES

[1] **Båge G.** and **Lindstrom G.**, *Committed Choice Functional Programming*, FGCS'88, Tokyo, Japan, Nov. 28 - Dec. 2, 1988, Vol. 2, pp. 666-674.

[2] **Barbuti R.**, **Bellia M.**, **Levi G.** and **Martelli M.**, *LEAF: a Language which Integrates Logic, Equations and Functions*, in [7], pp. 201-238.

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

[4] **Burton F. W.**, *Functional Programming for Concurrent and Distributed Computing*, The Computer Journal, Vol. 30, 5, 1987, pp. 437-450.

[5] **Clack C.** and **Peyton Jones S. L.**, *Strictness Analysis - a Practical Approach*, FPLCA'85, Nancy, France, Sept. 16-19, 1985, LNCS 201, pp. 35-49.

[6] **Clark K. L.** and **Gregory S.**, *PARLOG: A Parallel Logic Programming Language*, Research Report DOC 83/5, Imperial College, UK, 1983.

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

[8] **Dershowitz N.** and **Plaisted D. A.**, *Logic Programming cum Applicative Programming*, IEEE International Symposium on Logic Programming, Boston, USA, July 15-18, 1985, pp. 54-67.

[9] **Glauert J. R. W.**, **Hammond K.**, **Kennaway J. R.** and **Papadopoulos G. A.**, *Using Dactl to Implement Declarative Languages*, CONPAR'88, Manchester, UK, Sept. 12-16, 1988, Vol. 1, pp. 89-94

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

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

[12] **Hansson Å.**, **Haridi S.** and **Tärnlund S-Å.**, *Properties of a Logic Programming Language*, in Logic Programming, Clark K. L. and Tärnlund A. (eds), Academic Press, 1982, pp. 267-280.

[13] **Hudak P.** and **Smith L.**, *Para-Functional Programming: a Paradigm for Programming Multiprocessor Systems*, 13th Annual ACM Symposium on the Principles of Programming Languages, Florida, USA, Jan. 13-15, 1986, pp. 243 254.

[14] **Josephs M. B.**, *Functional Programming with Side-Effects*, Science of Computer Programming 7, 1986, North-Holland, pp. 279-296.

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

[16] **Kornfeld W. A.**, *Equality for Prolog*, in [7], pp. 279-294.

[17] **Papadopoulos G. A.**, *A Fine Grain Parallel Implementation of PARLOG*, TAPSOFT'89, Barcelona, Spain, Mar. 13-19, 1989, LNCS 352, Springer Verlag, pp. 313–327.

[18] **Robinson J. A.** and **Sibert E. E.**, *LOGLISP: Motivations, Design and Implementation*, in Logic Programming, Clark K. L. and Tärnlund A. (eds), Academic Press, 1982, pp. 299-314.

[19] **Sato M.** and **Sakurai T.**, *QUTE: a Functional Language Based on Unification*, in [7], pp. 131-156.

[20] **Subrahmanyam P. A.** and **You J-H.**, *FUNLOG: a Computational Model Integrating Logic Programming and Functional Programming*, in [7], pp. 157-198.

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

[22] **Papadopoulos G. A.**, *Parallel Implementation of Concurrent Logic Languages Using Graph Rewriting Techniques*, Ph.D. Thesis, University of East Anglia, Norwich, UK, December 1989.