Journal of Programming Languages **5**, 1997, 109–133

Implementing concurrent logic and functional languages in Dactl

George A. Papadopoulos

Department of Computer Science, University of Cyprus, 75 Kallipoleos Street, POB 537, CY-1678, Nicosia, Cyprus
george@turing.cs.ucy.ac.cy

A methodology is developed for mapping a wide class of concurrent logic languages (CLLs) onto Dactl, a compiler target language based on generalized graph rewriting. We show how features particular to the generalized graph rewriting model (such as non-root overwrites and sharing) can be used to implement CLLs. We identify problems in the mapping of a concurrent logic program to an equivalent set of rewrite rules and provide solutions. We also show some important optimizations and compilation techniques that can be adopted in the process. Finally, we take advantage of the underlying graph reduction model to enhance a concurrent logic program with some capabilities found usually only in functional languages such as lazy evaluation, sharing of computation and higher order programming.

Keywords: concurrent logic languages, term graph rewriting, functional programming, intermediate (compiler target) languages, language embeddings

The formalisms of graph rewriting (reduction) and logic programming have developed quite independently of each other. With the exception of a few cases, advances in the former have not been exploited by the latter and vice versa. In this paper we try to bridge the gap between the two formalisms by showing how concurrent logic languages can be implemented using graph rewriting. In particular, we develop techniques for mapping a wide class of CLLs including Parlog, GHC, Strand, Janus and a restricted subset of the Concurrent Prolog family onto Dactl, a compiler target language based on graph rewriting. We discuss the problems found in the process and the adopted solutions. The paper contributes to related research by:

- examining the potential of graph reduction as a suitable model for implementing CLLs in terms of expressiveness and efficiency
- showing that CLLs can be thought of as instances of the generalized graph rewriting model
- giving meaning to some properties of graph rewriting (such as sharing and arc redirection) in the logic programming world, thus identifying relationships between operations in graphs and 'equivalent' logic programs
- examining the potential of the intermediate language approach in general and Dactl in particular as an appropriate model for implementing concurrent logic languages
- taking advantage of the intermediate level of computation offered by Dactl in enhancing concurrent logic languages with some features found only in some other languages also mapped onto Dactl, and in particular functional ones.

The paper is organized as follows: section 1 introduces Dactl as an intermediate language in the context of rewriting systems; section 2 introduces concurrent logic languages. These are essentially condensed descriptions and the reader is referred to more specialized papers cited in the references for further details. Section 3 presents the mapping of concurrent logic programs onto equivalent sets of Dactl rewrite rules, and section 4 describes the enhancement of concurrent logic programs with functional capabilities. The paper ends with a short description of related work and some conclusions and directions for further research.

1 Dactl and the 'intermediate language approach'

Rewriting systems (Barendregt et al. 1987) offer a powerful computational model for declarative languages. It can be shown that a functional program can be mapped onto an equivalent canonical rewriting system. However, logic programs can also be seen as sets of equivalence preserving rewrite rules (Dershowitz 1985). It follows that a language based on rewriting theory has the potential of being an intermediate language (an UNCOL; (Steel 1961)) for a number of declarative languages. This decoupling of language design from actual implementation allows the easier evolution of languages as well as architectures.

Dactl is such an intermediate language. It was developed as part of the Alvey Flagship research program (Procter and Skelton 1988) to play the role of a compiler target language through which declarative languages would be implemented on fine grain parallel architectures such as Flagship (Watson et al. 1988), Alice (Darlington and Reeve 1981), GRIP (Peyton Jones et al. 1987), etc. In addition to the family of languages described in this paper, it has been shown that functional languages such as Hope+, ML, Common Lisp, Clean, etc. can be easily mapped onto Dactl rewrite rules (Darlington et al. 1988, Hammond 1990, Kennaway 1988). A subset of Dactl has been implemented on the Flagship machine (Banach and Watson 1988); an ongoing implementation was also reported for the GRIP machine (Peyton Jones et al. 1987) while Hammond (1990) studies a possible implementation on transputers. There is also an interpreter running under Unix on Suns and on Macintoshes (Glauert et al. 1988a). In this paper, however, we will not consider the implementation of Dactl; instead we will concentrate on the high level transformations necessary to derive a Dactl program operationally equivalent to the original concurrent logic program.

A Dactl program is a set of rewrite rules specifying possible transformations of graph objects. In addition, a set of control markings is used to specify the required reduction stategy, i.e. the choice procedure for selecting candidate redexes from those available in the graph. Following Kowalski's well-known equation 'Algorithm = Logic + Control', the rules comprise the Logic part and the control markings the Control part. The reason for decoupling the logic from control in a Dactl program at the programmer's level is the need to accommodate different families of languages with possibly divergent operational semantics (lazy functional languages, 'eager' concurrent logic languages) for which no general predefined reduction strategy is adequate.

The following definition of an append function in Dactl illustrates the above points:

$$Append[Nil\ y] \Rightarrow *y;$$
$$Append[Cons[h\ t]\ y] \Rightarrow \#Cons[h\ \hat{}\ *Append[t\ y]];$$

Similar notation is used both for rewritable functions (such as *Append*) and for constructions (such as *Cons* or *Nil*); however, there will be no rules for rewriting *Cons* or *Nil* nodes. Each node has a symbol and

a list of arcs to successor nodes. The first rule specifies that an *Append* node with *Nil* as first argument is overwritten with the second argument *y*. The activation marking '*' will activate *y*, causing further evaluation if the latter is a rewritable function. The second rule is applicable when the first argument of *Append* is a *Cons*, in which case *Append* is overwritten to a new *Cons* node bearing the suspension marking '#' whose second argument is a recursive call to *Append*. This call is activated using '*', and the notification marking '^' 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 fully evaluated. To illustrate Dactl's flexibility in accommodating different operational semantics, consider the following variations of *Append*'s second rule:

$$Append[Cons[h\ t]\ y] \Rightarrow *Cons[h\ *Append[t\ y]];$$
$$Append[Cons[h\ t]\ y] \Rightarrow *Cons[h\ Append[t\ y]];$$

The first rule specifies an eager evaluation strategy where the partial result of *Append*'s reduction is made available to its caller while the recursive call is executed in parallel. The second rule shows a lazy version; the recursive *Append* will remain dormant until the original caller activates it again. A more elaborate description of Dactl can be found in the accompanying paper (Glauert et al. 1997).

2 Concurrent logic languages

Concurrent logic languages (CLLs) comprise a family of languages based on a subset of first order predicate calculus (Horn clauses) and utilizing stream parallelism. They provide a powerful computational model able to model reactive systems and easily amenable to parallel execution. A concurrent logic program is a set of guarded Horn clauses of the form

$$H : -G1, \ldots, Gm \mid B1, \ldots, Bn \quad m, n \geq 0$$

where H is the head, $\mid$ is the commit operator, $G1, \ldots, Gm$ is the guard part and $B1, \ldots, Bn$ is the body part. Declaratively, the meaning of the above clause is that H is true if both $G1, \ldots, Gm$ and $B1, \ldots, Bn$ are true. Operationally, the guard calls $G1$ to Gm are evaluated first in parallel and upon successful termination the computation commits to the body of the clause. The head H is of the form $p(t1, \ldots, tn)$ where p/n is a predicate name of arity n and $t1, \ldots, tn$ are its arguments. There may be more than one rule with the same name p and arity n, in which case they form a group definition of the process p.

The computation starts with a set of cooperating processes (goals) executing in parallel and communicating by means of shared variables. The clauses of a program specify the behaviour and the various transitions possible for each process. If for a certain goal to be reduced there is more than one candidate clauses to select from, the first one to solve its guard successfully will be chosen and the computations in the guards of the other candidate clauses will be abandoned. Thus CLLs incorporate the concept of committed choice 'don't care' non-determinism from CSP.

The variables shared among the processes form communication channels; some of the processes in a computation (termed the **producers**) are allowed to send messages over these channels (by means of **output unification**) while others are waiting for messages (**input unification**). A number of CLLs have been developed over the years that differentiate in the synchronization mechanism they employ for managing the processes running in parallel, ranging from compile time based input/output data flow specification

(Parlog, GHC, Fleng, Janus) to run time based read only unification (Concurrent Prolog) or determinacy conditions (P-Prolog).

The usual way to implement concurrent logic languages is by means of variants of the highly successful WAM (Warren 1983, Aït-Kaci 1991). In particular, in a parallel implementation of a CLL, each processor in the system executes an emulator of the abstract machine. The latter comprises a reduction component and a communication component. The reduction component is responsible for local process scheduling and execution and supports the basic operations required by the operational model (reading and writing of data, process management). The communication component is responsible for managing the messages generated to access remote variables (those residing on other processors) for reading or writing. In this paper we offer an alternative way to implement a CLL based on a high-level intermediate language approach.

Concurrent logic languages simplify considerably the programming of parallel machines (Foster and Taylor 1990, Tick 1991) and they formed the basis on which the Japanese Fifth Generation Computer System (FGCS) project was implemented (Furukawa 1991). A comprehensive survey of the development of CLLs can be found in Shapiro (1989).

3 Translating concurrent logic programs into Dactl rewrite rules

3.1 Principles of the translation

We recall that a concurrent logic program is a set of axioms (clauses) defining relationships between objects. By 'objects' we refer to terms manipulated by the clauses using unification. The way unification is used to manipulate terms is defined by an associated execution strategy which is based on stream parallelism but is not necessarily identical for all languages (there may be differences, for instance, in the search operators employed or the exact kind of synchronization used). However, it is possible to associate all these languages with a process interpretation which involves, among others, mapping of goals to processes, and synchronization and communication between the latter two. In implementing a concurrent logic language in Dactl, therefore, the following main issues arise: representation of terms, support for the unification mechanism employed, and the ability to express adequately the language's process interpretation in the underlying implementation model.

Regarding terms, a number or a string preserves its usual representation in Dactl; in addition, special patterns are provided (*INT*, *REAL*, etc.) to allow testing of nodes matched in the left-hand side of rules. A list $[H \mid T]$ is represented as $Cons[h\ t]$ with *Nil* being the empty list, and any other data structure $f(t1, \ldots, tn)$ is represented as $Tup['f't1 \ldots tn]$.

The exact way unification is implemented varies from one language to the other depending on the language's semantics; the same is true about the representation of logic variables (as we will see below). However, any unification procedure includes rules for instantiating variables and decomposing data structures. In particular, a Dactl unification function has some variable instantiation rule

$$Unify[v : Var\ term] \rightarrow v := *term;$$

and some data decomposition rules like the following for lists:

$$Unify[Cons[h1\ t1]\ Cons[h2\ t2]] \rightarrow *Unify[h1\ h2], *Unify[t1\ t2];$$

Note that in the first rule we have fired *term*; this will activate any process suspending on *v* waiting for its instantiation. The principles of mapping a concurrent logic program onto a set of Dactl rewrite rules are illustrated by means of the following concurrent logic version of *append*:

$$append([H \mid T], Y, Z) : -true \mid Z = [H \mid Z1], append(T, Y, Z1).$$
$$append([\,], Y, Z) : -true \mid Z = Y.$$

A possible translation to Dactl is the following:

$$Append[Cons[h\ t]\ y\ z] \Rightarrow *Append[t\ y\ z1 : Var], *Unify[z\ Cons[h\ z1]];$$
$$Append[Nil\ y\ z] \Rightarrow *Unify[z\ y];$$
$$Append[l : Var\ y\ z] \Rightarrow \#Append[\hat{}l\ y\ z];$$
$$Append[ANY\ ANY\ ANY] \Rightarrow *FAIL;$$

The first two rules perform the reduction as defined by the original append program. Note here the introduction of the new node $z1$ with the pattern *Var*; Dactl does not support 'open graphs' with true variables. This turns out to be an advantage, however, as we will see later on. The third rule models the required synchronization mechanism and in particular the suspension of the process until its first argument is instantiated to some value. The last one reports failure. This last rule is not always necessary, depending on how each language treats the notion of failure. In general, a procedure in a concurrent logic program is translated into an equivalent Dactl rule set comprising a sufficient number of rewrite rules to perform the required unification and execution of guard and body calls, followed by a suspension rule and a failure rule.

Depending on the type of head unification that must be performed and the existence or not of guards we have basically three types of procedures:

- unguarded
- guarded with non-overlapping patterns
- guarded with overlapping patterns.

The above procedure belongs to the first category. The translation to Dactl of this first category is quite trivial since the language's semantics captures completely the needed functionality.

3.2 Translating languages with non-atomic unification

The group of languages with non-atomic unification includes (among others) Fleng (Nilsson and Tanaka 1986), Parlog (Gregory 1987), GHC (Ueda 1986), Strand (Foster and Taylor 1990) and Janus (Saraswat et al. 1990). Consider the definition of the predicate $member\_add(X, L, Lb, Le)$, adding an element X into the difference list Lb/Le if it is a member of the list L (Foster and Taylor 1990):

$$member\_add(X, [X1 \mid \_], Lb, Le) : -X == X1 \mid Lb = [X \mid Le].$$
$$member\_add(X, [X1 \mid L], Lb, Le) : -X = \backslash = X1 \mid member\_add(X, L, Lb, Le).$$
$$member\_add(\_, [\,], Lb, Le) : -Lb = Le.$$

This example belongs to the category of procedures having guarded clauses with non-overlapping patterns. The translation to Dactl optimizes the (identical) input unification performed by the first two clauses:

$$
\begin{aligned}
&\mathit{Member\_add}[x\ \mathit{Cons}[x1\ l]\ lb\ le] \Rightarrow \#\mathit{Member\_add\_Commit}[\hat{}g1\ \hat{}g2\ xl\ lb\ le],\\
&\qquad\qquad g1 : *\mathit{Eq}[x\ x1], g2 : *\mathit{NotEq}[x\ x1];\\
&\mathit{Member\_add}[\mathit{ANY\ Nil}\ lb\ le] \Rightarrow *\mathit{Unify}[lb\ le];\\
&\mathit{Member\_add}[x\ l : \mathit{Var}\ lb\ le] \Rightarrow \#\mathit{Member\_add}[x\ \hat{}l\ lb\ le];\\
&\mathit{Member\_add}[\mathit{ANY\ ANY\ ANY\ ANY}] \Rightarrow *\mathit{FAIL};\\
&\mathit{Member\_add\_Commit}[\mathit{SUCCEED\ ANY}\ x\ l\ lb\ le] \Rightarrow *\mathit{Unify}[lb\ \mathit{Cons}[x\ le]];\\
&\mathit{Member\_add\_Commit}[\mathit{ANY\ SUCCEED}\ x\ l\ lb\ le] \Rightarrow *\mathit{Member\_add\_Commit}[x\ l\ lb\ le];\\
&\mathit{Member\_add\_Commit}[\mathit{FAIL\ FAIL\ ANY\ ANY\ ANY\ ANY}] \Rightarrow *\mathit{FAIL};\\
&r : \mathit{Member\_add\_Commit}[\mathit{ANY\ ANY\ ANY\ ANY\ ANY\ ANY}] \rightarrow \#r;
\end{aligned}
$$

The first rule performs the unification required for the first two clauses and calls *Member_add_Commit* to solve the guards. The rest of the rules can be understood easily. The last rule of *Member_add_Commit* is a bit tricky, however; it will suspend the root packet waiting for further signals from any remaining child processes. Note that *Member_add_Commit* refrains from building the body calls if they are not needed; this is similar to the lazy creation technique used in implementing functional languages. Note also that a sophisticated compiler would exploit the complementary nature of $==$ and $=\backslash=$ and produce code for only the first test.

The above program was written in (kernel) Parlog and adheres to Parlog semantics: bindings to variables are created by full output unification and failure to match during input unification must be reported. (Safe) GHC imposes the same constraints and the translation to Dactl of the corresponding GHC program is identical to the one shown above. Strand, however, uses assignment instead of unification. In addition, the trapping of failure is the responsibility of the programmer rather than the underlying implementation model. To translate Strand programs, therefore, there is no need for either a failure rule or a call to some Unify Dactl function. Assuming Strand semantics, *member_add* would be translated to Dactl as follows:

$$
\begin{aligned}
&\mathit{Member\_add}[x\ \mathit{Cons}[x1\ l]\ lb\ le] \rightarrow \#\mathit{Member\_add\_Commit}[\hat{}*\mathit{Eq}[x\ x1]\ xl\ lb\ le];\\
&\mathit{Member\_add}[\mathit{ANY\ Nil}\ lb : \mathit{Var}\ le] \rightarrow lb := *le;\\
&\mathit{Member\_add}[x\ l : \mathit{Var}\ lb\ le] \rightarrow \#\mathit{Member\_add}[x\ \hat{}l\ lb\ le];\\
&\mathit{Member\_add\_Commit}[\mathit{SUCCEED}\ x\ l\ lb : \mathit{Var}\ le] \rightarrow lb := *\mathit{Cons}[x\ le];\\
&\mathit{Member\_add\_Commit}[\mathit{FAIL}\ x\ l\ lb\ le] \rightarrow *\mathit{Member\_add\_Commit}[x\ l\ lb\ le];\\
&r : \mathit{Member\_add\_Commit}[\mathit{ANY\ ANY\ ANY\ ANY\ ANY}] \rightarrow \#r;
\end{aligned}
$$

Here we have performed the optimization regarding the guard tests that was mentioned above. Note that Strand's assignment can be modelled directly in Dactl by means of non-root overwrites. The reason for dispensing with the need for a full unification primitive stems from the language's (reasonable) restriction that only one producer process should exist for each variable. The language Janus, developed for distributed concurrent logic programming, goes even further with respect to this restriction and imposes the additional constraint that only one consumer should exist for each variable. One of the main advantages of

this restriction is the simplification of memory reclamation. The equivalent Janus program for the above example is :

$$\begin{aligned} member\_add(X,[X1 \mid L],!Lb,Le) :: & X == X1 \rightarrow Lb = [X \mid Le], \\ & X = \backslash = X1 \rightarrow member\_add(X,L,!Lb,Le). \\ member\_add(\_,[],!Lb,Le) :: & Lb = Le. \end{aligned}$$

where ! denotes the occurrence of the variable that can be written. The translation of the above program in Dactl is the same as for the Strand version but it is expected that the underlying implementation would take advantage of the single-producer single-consumer constraint and reuse immediately the data nodes matched in the left-hand side of a rule (such as *Cons* and *Nil* above). Here the generated Dactl program could be enhanced with appropriate annotations on those data nodes that can be reused to assist the translator of Dactl programs to native machine code (such as King and Glauert (1991)). Such nodes would be overwritten directly with the new data instead of using indirection nodes. Note that these optimizations can be performed for any concurrent logic program, not necessarily written in Janus, that has this property something that can be checked by means of compile time or run time analysis (Foster and Winsborough 1991).

Note also that there is no need to perform root overwrites (i.e. rewrites) of Dactl functions corresponding to Strand or Janus predicates using $\Rightarrow$. This is a point that raises some interesting comparison issues in the way Strand or Janus and the rest of the CLLs are mapped onto a graph rewriting model. In particular, consider a conjunction of goals $g1(\ldots),\ldots,gn(\ldots)$ written in a language that should detect failure , such as Parlog or GHC. The equivalent set of Dactl functions executing in parallel should be monitored by an *AND* function which would detect failure as soon as possible:

$$LHS \Rightarrow \#AND[\hat{}*G1[\ldots]\ldots\hat{}*Gn[\ldots]];$$

$$AND[SUCCEED\ldots SUCCEED] \Rightarrow SUCCEED;$$
$$r : AND[(ANY - FAIL)\ldots(ANY - FAIL)] \rightarrow \#r;$$
$$AND[ANY\ldots ANY] \Rightarrow *FAIL;$$

Any predicate *G* would have to be mapped to either *SUCCEED* or *FAIL* by means of suitable rules as shown above. For the case of Strand or Janus, however, this is not necessary and we can dispense with these extra rules and monitoring *AND* functions. It suffices to simply spawn the processes in the new graph constructed after the successful matching with the left-hand side of the rule:

$$LHS \rightarrow *G1[\ldots],\ldots,*Gn[\ldots];$$

The use of $\rightarrow$ instead of $\Rightarrow$ states that the right-hand side of the rule will spawn some new processes (*G*1 to *Gn*) but will not change the state of the left-hand side function (because it is irrelevant to the computation). Any communication will be done by means of shared overwritable nodes. It is worth pointing out that the functionality of $\rightarrow$ is not easily expressible in a CLL (although Janus semantics come closer to this notion than those of any other CLL).

As a final point on implementing Janus in Dactl, we note that the language supports arrays. These can be easily implemented in Dactl using the latter's support for vector operations (Glauert et al. 1988a).

We recall that there are three categories of procedures in the translation of concurrent logic programs to Dactl, and so far we have considered only the first two. The efficient translation of procedures belonging to the third category is not a trivial issue since all candidate clauses must be evaluated in parallel. The techniques reported in the literature generally involve a lot of copying (Gregory 1987, Levy and Shapiro 1987). Our technique (described fully in Papadopoulos (1989a), Papadopoulos (1989b) involves the use of an overwritable node to be instantiated to the body of the committed clause. In particular, assuming a procedure of the form

$$p(\ldots) : -g1(\ldots) \mid b1(\ldots).$$
$$p(\ldots) : -g2(\ldots) \mid b2(\ldots).$$

where the input patterns of p are overlapping and $g1$,$g2$ are non-flat guards, the translation to Dactl adheres to the following pattern:

$$P[\ldots] \Rightarrow commit : Var, \#\#OR[\hat{}\, o1\ \hat{}\, o2\ commit],$$
$$o1 : *P[1 \ldots commit], o2 : *P[2 \ldots commit]$$

$$OR[FAIL\ FAIL\ commit : Var] \rightarrow commit := *FAIL;$$

$$P'[1 \ldots commit] \Rightarrow *Eval\_G1[commit];$$
$$P'[2 \ldots commit] \Rightarrow *Eval\_G2[commit];$$

$$Eval\_G1[SUCCEED \ldots commit : Var] \rightarrow commit := *B1[\ldots];$$
$$Eval\_G2[SUCCEED \ldots commit : Var] \rightarrow commit := *B2[\ldots];$$

Note that the input unification performed is absorbed completely by the Dactl model and involves no copying. Overwritable nodes, like the *commit* above, that are instantiated to functions rather than constructors, play the role of **metavariables** and in the graph rewriting world are referred to as **stateholders**.

We now turn our attention to the optimization of the suspension patterns. There is an interesting similarity here with the way functional languages, and in particular lazy ones, are compiled to Dactl (Kennaway 1988, Hammond 1990). There, a number of firing rules are necessary to activate the inner function applications as detected by strictness analysis. Here, we need the opposite – a suspension rule that will detect any unistantiated input arguments and suspend on them until they get their values from their 'eager' producers running in parallel. Consider the following Parlog or GHC procedure:

$$P(1,2,3).$$
$$p(4,\_,5).$$
$$p(\_,\_,6).$$
$$p(\_,\_,7).$$

Using the techniques described so far, the translation to Dactl is as follows:

$$P[1\ 2\ 3] \Rightarrow *SUCCEED;$$
$$P[4\ ANY\ 5] \Rightarrow *SUCCEED;$$
$$P[ANY\ ANY\ 6] \Rightarrow *SUCCEED;$$

$$P[\mathit{ANY\ ANY}\ 7] \Rightarrow *\mathit{SUCCEED};$$
$$(P[p1\ p2\ p3]\&(P[(\mathit{Var}+1)(\mathit{Var}+2)(\mathit{Var}+3)]+P[(\mathit{Var}+4)\ \mathit{ANY}\ (\mathit{Var}+5)]$$
$$+(P[\mathit{ANY\ ANY}(\mathit{Var}+6)]+P[\mathit{ANY\ ANY}(\mathit{Var}+7)])) \Rightarrow \#P[\hat{}p1\ \hat{}p2\ \hat{}p3];$$
$$P[\mathit{ANY\ ANY\ ANY}] \Rightarrow *\mathit{FAIL};$$

Note the use of the pattern operators available in Dactl to express the conditions under which the process *P* should suspend. The left-hand side of the rule comprises a number of different pattern combinations, all causing suspension of the process, unioned together. However, the suspension rule is naive and unnecessarily complex since in many cases it is possible to collapse a number of patterns into one general one. A number of techniques have been devised to optimize suspension patterns and these are summarized by the following rules. These techniques can also be used in implementing other applicative languages using a pattern matching language like Dactl.

Rule 1 If the matching pattern of a clause has only one non-variable term *Term*, then the corresponding position in the suspension pattern for that clause is required to match *Var* only rather than $(\mathit{Var}+\mathit{Term})$.

This is justified because had the position been *Term* it would have matched the corresponding matching rule. The suspension rule now becomes :

$$(P[p1\ p2\ p3]\&(P[(\mathit{Var}+1)(\mathit{Var}+2)(\mathit{Var}+3)]+P[(\mathit{Var}+4)\ \mathit{ANY}\ (\mathit{Var}+5)]$$
$$+(P[\mathit{ANY\ ANY\ Var}]+P[\mathit{ANY\ ANY\ Var}])) \Rightarrow \#P[\hat{}p1\ \hat{}p2\ \hat{}p3];$$

Rule 2 After applying the first rule to some position *i* in a suspension pattern, then in all the other suspension patterns that expect a non-variable term *Term* in *i*, that position is required to match *Var* only rather than $(\mathit{Var}+\mathit{Term})$.

This is justified since the case of positions *i* having *Var* is now covered by the patterns to which the first rule has been applied. The suspension rule now becomes:

$$(P[p1\ p2\ p3]\&(P[(\mathit{Var}+1)\ (\mathit{Var}+2)\ 3]+P[(\mathit{Var}+4)\ \mathit{ANY}\ 5]$$
$$+(P[\mathit{ANY\ ANY\ Var}]+P[\mathit{ANY\ ANY\ Var}])) \Rightarrow \#P[\hat{}p1\ \hat{}p2\ \hat{}p3];$$

Rule 3 After applying rules 1 and 2 where possible, if there are any suspension patterns with a single expression $(\mathit{Var}+\mathit{Term})$ in any position, this expression is simplified to *Var*.

This is justified for reasons similar to the ones for rule 1. The suspension rule now becomes :

$$(P[p1\ p2\ p3]\&(P[(\mathit{Var}+1)\ (\mathit{Var}+2)\ 3]+P[\mathit{Var\ ANY}\ 5]+(P[\mathit{ANY\ ANY\ Var}]$$
$$+P[\mathit{ANY\ ANY\ Var}])) \Rightarrow \#P[\hat{}p1\ \hat{}p2\ \hat{}p3];$$

Rule 4 After applying any of the first three rules, if there exist any repeated occurrences of a suspension pattern, these are eliminated.

This is self-explanatory. Our suspension rule takes its final form:

$$(P[p1\ p2\ p3]\&(P[(\mathit{Var}+1)\ (\mathit{Var}+2)\ 3]+P[(\mathit{Var}+4)\ \mathit{ANY}\ 5]$$
$$+(P[\mathit{ANY\ ANY\ Var}])) \Rightarrow \#P[\hat{}p1\ \hat{}p2\hat{}\ p3];$$

Rule 5 After applying any of the first four rules, if there are any suspension patterns that have no occurrences of *Var*, these patterns are eliminated.

This is justified on the grounds that such a pattern would be identical to the one in the corresponding matching rule. Sometimes, after applying the above transformations, such patterns are generated; the last rule eliminates any of these patterns.

Deep pattern matching will not be discussed here. There are a number of issues related to how patterns involving deep data structures should be matched: they focus mainly on the tradeoff between performing the matching operations efficiently and preserving order independence semantics. Most implementations of concurrent logic languages prefer to perform deep pattern matching sequentially, thus improving the performance of these operations at the expense of compromising the order independence nature of proper unification. In Papadopoulos (1989a) we describe an algorithm for an order independent compilation of deep patterns and its implementation in Dactl. There, we also discuss the implementation of other features of CLLs such as metacalls.

The techniques discussed so far can be used to implement in Dactl all CLLs having non-atomic unification, with the exception of the unsafe subset of GHC. For that, a run time safety check is required that will suspend all instantiations of a variable outside its own environment. In Glauert and Papadopoulos (1988) we describe an elegant technique for performing this run time safety test in Dactl. It involves extending the notion of a GHC variable to include its birthplace. In particular, a GHC variable is now represented as $Var[env]$ where *env* is a pointer to the place it was created. The run time test can therefore be implemented as a simple pointer equality test with unification rules of the following sort:

$$Run\_Time\_Unify[env\ v : Var[env]\ term] \Rightarrow *SUCCEED, v := *term;$$
$$Run\_Time\_Unify[env1\ v : Var[env2]\ term] \Rightarrow \#Unify[\hat{}v\ term];$$

We recall here that in the graph rewriting world a repeated occurrence of a variable denotes sharing of a subgraph. Note that the hierarchical nature of local environments in GHC guarantees that the run time test will be performed at most once for each variable. Hence the use of the ordinary *Unify* function in the second rule above. Dactl's flexibility in defining what constitutes a variable will be further exploited later on.

We end this section with some performance analysis. Table 1 presents some performance figures comparing the efficiency of the Dactl programs produced by our Parlog and GHC to Dactl compilers with the original versions running under the SPM system for Parlog (Gregory et al. 1989) and the Logix system for FCP (Silverman et al. 1988). All systems were installed on a Sun 3/180S running Sun OS 3.5. Time was measured using the relevant facilities of the C-based Dactl interpreter and the Logix system, and the Unix facilities for the case of SPM. Rewrites (or reductions) (R) refers to the number of process reductions performed by the system, parallel cycles (PC) refers to the number of emulator cycles performed and reductions per cycle (RPC) refers to the average and maximum number of rewrites performed per emulator cycle. For the last two parameters we assume the system having available an infinite number of processors. Note that the SPM system has no statistics facilities available. Note also that we do not claim any relationship between the relevant parameters of the Dactl interpreter and the Logix system but present them simply as a profile of the systems' behaviour.

Bearing in mind that the Dactl interpeter does not enjoy the compiler technology that was developed for the Logix and SPM systems, we note that Dactl programs that perform essentially symbolic manipulations (such as append, merge or binary tree) produce comparable results regarding execution time to

Table 1: Performance figures: Dactl interpreter, Logix and SPM

System	Dactl interpreter				Logix				SPM
Benchmarks	T	R	PC	RPC	T	R	PC	RPC	T
append (100+0)	0.70	509	315	2.91 7	0.60	1041	202	5	0.26
merge (100+100)	1.2	1012	615	2.96 7	1.0	1641	302	5	0.3
primes (2 to 300)	100	53 726	4495	18.25 39	20	16 969	1166	14	11
quicksort (100 els not rev.)	16	13 759	912	24.04 79	7	7218	258	27	1.5
binary tree (depth 5)	0.18	157	46	5.15 17	0.24	259	24	10	0.2

T, time (s); R, rewrites (reductions); PC, parallel cycles; RPC, rewrites (reductions) per cycle (the two values for the Dactl interpeter correspond to the mean and peak value respectively).

those produced by the other two systems. The significant differences in execution time for the other two programs (primes, quicksort) may be attributed to how arithmetic and other operations are implemented in each system. A further indication of the validity of the above argument is the significant increase in rewrites (R) performed by the Dactl system for the latter two programs when compared with the other three. Finally, the consistently higher number of emulator cycles (PC) performed by the Dactl system, as compared to those performed by Logix, can be attributed to the inherently finer grain nature of a Dactl program, when compared to an equivalent Flat Concurrent Prolog one.

Tables 2 and 3 compare the performance of the Dactl programs produced by our Parlog/GHC compilers with the similar ones produced by the Clean to Dactl (Kennaway 1988) and ML to Dactl (Hammond 1990) compilers. The first example is a typical deterministic program with plenty of **horizontal** parallelism and the second is a typical non-deterministic program with a high degree of speculative parallelism. The results show that in the first case the functional versions generated by the Dactl compilers are oversequential while in the second case the functional versions overcompute. The ability of the concurrent logic versions to kill any remaining tasks after commitment (using techniques described in Papadopoulos (1989a) that are similar to the short circuit one) renders the latter more efficient. More to the point, Table 2 shows that the CLL version performs less parallel cycles than the other two versions while at the same time exhibiting a higher degree of parallelism. Table 3 shows that if only part of the speculative computation need be performed (as in the case for finding the seventh element of the binary tree) the CLL version will manage to stop execution before completing the whole search, thus avoiding performing unneccesary computations (only 92 rewrites will be performed compared with 221 for the Clean and 197 for the ML versions). Only when the whole binary tree must be explored do the three versions exhibit similar results, where in fact the ML version turns out to be the faster of all. This is attributed to the fact that in this case there is little benefit in searching the tree completely in parallel; the higher parallelism of the CLL version (compared to that of the Clean or ML programs) introduces some extra overhead without providing any benefits (as in the first case where unneccesary computations are killed). A broader comparison between the various language models using Dactl as common reference basis is reported in Hammond and Papadopoulos (1988), Glauert et al. (1988b).

Table 2: Performance comparison: quicksort

Program	Quicksort (50 elements reversed)			
Langs	**R**	**PC**	**AvP**	**MxP**
CLL	10 250	836	19.95	46
Clean	6654	11 357	1.11	2
ML	19 879	9081	1.78	3

CLL, Parlog/GHC; R, rewrites; PC, parallel cycles performed; AvP, activations processed per cycle (mean value); MxP, activations processed per cycle (peak value).

Table 3: Performance comparisons: tree search

Programs	Tree search (finding the 7th element)				Tree search (finding the 31st element)			
Langs	**R**	**PC**	**AvP**	**MxP**	**R**	**PC**	**AvP**	**MxP**
CLL	92	37	3.81	11	278	56	7.41	33
Clean	221	63	6.32	24	275	63	7.84	30
ML	197	52	5.69	24	245	52	7.08	30

CLL, Parlog/GHC; R, rewrites; PC, parallel cycles performed; AvP, activations processed per cycle (mean value); MxP, activations processed per cycle (peak value).

3.3 Translating languages with atomic unification

We turn our attention now to those CLLs that have atomic unification and we present the basic principles of mapping them to Dactl. In particular, we consider some members of the Concurrent Prolog family (Shapiro 1989), namely FCP(|), FCP(:), FCP(?) and FCP(:, |). The mappings presented here, unlike those of the previous section, rely on the full power of Dactl and in particular atomicity of rewrites. We recall from section 1 that all redirections indicated in the right-hand side of a rule are performed as a single action. This section therefore provides a justification of Dactl's insistence on atomicity and an interpretation of it in the world of CLLs based on atomic unification. Note that Dactl does not support full atomic unification; instead it offers the more limited form of atomic assignment. In most practical cases this suffices since at least one of the terms involved in the unification is a single variable. However, an atomic unification primitive can be supported by Dactl, without altering its semantics, based on transforming a unification into a set of atomic assignments. Although such a primitive can be implemented at a lower level for efficiency, it is imperative to be able to give it a meaning in the Dactl world. A top level definition in Dactl of such a primitive is as follows:

$$AtomicUnify[term1\ term2] \Rightarrow \#Merge\_Envs[\hat{}s], \quad s : *Select\_Args[term1\ term2];$$
$$Select\_Args[term1\ term2] \Rightarrow *P[p1 \ldots pn], \quad p1 : Pair[v1 : Var\ t1], \ldots pn : Pair[vn : Var\ tn];$$
$$Merge\_Envs[P[Pair[v1 : Var\ t1] \ldots Pair[vn : Var\ tn]]] \Rightarrow *SUCCEED, \quad v1 := *t1, , vn := *tn;$$
$$Merge\_Envs[pairs] \Rightarrow \#Merge\_Envs[\hat{}newpairs], \quad newpairs : *Select\_Args[pairs];$$

The implementation of the atomic unification primitive *Atomic_Unify* involves the use of the function *Select_Args* which collects all the variables involved in a unification as the first argument of the structure *Pair* and all the non-variable terms as its second. *Merge_Envs* is then invoked which uses Dactl's multiple non-root overwriting mechanism to instantiate all variables atomically. Note that *Select_Args*'s computation is not done atomically but in parallel with the rest of the activities in the graph. Thus it is possible for some of the variables involved in the unification to be instantiated by the time *Merge_Envs* is called. In that case the first rule of *Merge_Envs* will fail to match (since some *vi* no longer has the pattern *Var*) and the second rule will be invoked which will recompute the variable term pairs. This scheme is similar to the **eager broadcasting** one since it effectively allows the propagation of values made in the global environment into the local ones.

The use of the primitive *AtomicUnify* suffices for the mapping of FCP(|) to Dactl. FCP(|) differs from Parlog or GHC mainly in its ability to handle the short circuit technique in a more satisfactory way. Using *AtomicUnify* the compound unification $(X,L) = (Y,R)$ where $X = Y$ is the base computation and $L - R$ is the short circuit can be performed atomically. As an example, consider the top level part of an FCP(|) metainterpreter that allows interrupt handling, termination detection, and the computation of live and frozen snapshots (Shapiro 1989):

$$reduce(true, Is, L - R) : -L = R.$$
$$reduce(X = Y, Is, L - R) : -(X,L) = (Y,R).$$
$$reduce((A,B), Is, L - R) : -reduce(A, Is, L - M), reduce(B, Is, M - R).$$
$$reduce(goal(A), Is, L - R) : -clause(A, B, Is), reduce(B, Is, L - R).$$
$$reduce(A, [I \mid Is], L - R) : -serve\_interrupt([I \mid Is], A, L - R).$$

This is translated to Dactl as follows:

$$Reduce[True\ is\ l\ r] \Rightarrow *AtomicUnify[l\ r]$$
$$Reduce[AtomicUnify[x\ y]\ is\ l\ r] \Rightarrow *AtomicUnify[P[x\ l]\ P[y\ r]];$$
$$Reduce[G[a\ b]\ is\ l\ r] \Rightarrow \#AND[\hat{}b1\ \hat{}b2], \quad b1 : *Reduce[a\ is\ l\ m : Var], b2 : *Reduce[b\ is\ m\ r];$$
$$Reduce[Goal[a]\ is\ l\ r] \Rightarrow \#AND[\hat{}b1\ \hat{}b2], \quad b1 : *Clause[a\ b : Var\ is], b2 : *Reduce[b\ is\ l\ r];$$
$$Reduce[a\ s : Cons[i\ is]\ l\ r] \Rightarrow *Serve\_Interrupt[s\ a\ l\ r];$$
$$Reduce[p1 : Var\ p2\ p3\ p4] \Rightarrow \#Reduce[\hat{}p1\ \hat{}p2\ p3\ p4];$$
$$Reduce[ANY\ ANY\ ANY\ ANY] \Rightarrow *FAIL;$$

A final point worth making in the implementation of the short circuit technique in Dactl involves the use of repeated variables in the head of a clause. Consider the following base case in detecting the closing of a short circuit:

$$terminate(X - X, \ldots) : -report\ termination$$

Since the head of an FCP(|) clause is considered as being passive (*à la* Parlog or GHC), the indicated unification can be treated as equality checking. The translation to Dactl is trivial:

$$Terminate[x\ x\ \ldots] \Rightarrow report\ termination;$$

In fact here we illustrate another interpretation of pointer equality in graphs; pointers play the role of switches that link various computations around a graph. Once a computation completes execution it merges the corresponding arcs using redirection:

$$Process[\ldots\ l\ r] \Rightarrow \#Terminate[\hat{}*Compute[\ldots]\ l\ r];$$
$$Terminate[ANY\ l : Var\ r : Var] \rightarrow l := r;$$

(Note that there is no need to fire the node *r*; it suffices to collapse the graph nodes for each variable into a single node.) This then can be detected using pointer equality.

Another member of the Concurrent Prolog family is FCP(:). A program in FCP(:) is a set of clauses of the form

$$Head : -Ask : Tell \mid Body$$

where the *Tell* part is allowed to unify global variables before commitment. Since upon failure such an attempted unification should leave no trace, it must be performed as an atomic action. Dactl's multiple redirection facility combined with pattern matching offers a limited form of ask–tell unification. Consider the following example:

$$Test\_Set\_Spawn[v1 : Var\ v2 : Var\ Cons[p1\ p2]] \Rightarrow *SUCCEED, v1 := *1, v2 := *2, *Pr1[p1], *Pr2[p2];$$
$$Test\_Set\_Spawn[v1\ v2\ l : Var] \Rightarrow \#Test\_Set\_Spawn[v1\ v2\ \hat{}l];$$
$$Test\_Set\_Spawn[ANY\ ANY\ ANY] \Rightarrow *FAIL;$$

The process *Test_Set_Spawn* will suspend until its third argument is instantiated to a list. It will then attempt to instantiate $v1$ and $v2$ and spawn $Pr1$ and $Pr2$. If either of $v1$ or $v2$ is not a variable or the third argument is not a list, the third rule will be chosen. Note that the first rule performs both ask unification (matching of the *Cons* structure) and tell unification (redirection of the two variables), and both kinds of unification are performed before commitment. For if either the ask or tell part fails (l is not a list or $v1$ or $v2$ is not a variable), failure to select the first rule would leave no trace.

In most cases this limited form of tell unification suffices; for the very few cases where full unification is needed, the primitive *AtomicUnify* can be used. As an example consider the following FCP(:) program that simulates CSP with output guards (Shapiro 1989):

$$p(X, ToC1, ToC2) : -ToC1 = hello(X), ToC2 = hello(X).$$

$$c(Id, hello(X1), \_) : -true : X1 = Id \mid true.$$
$$c(Id, \_, hello(X2)) : -true : X2 = Id \mid true.$$

where the intention is that c should non-deterministically and atomically select a variable ($X1$ or $X2$) and bind it to its unique id. This program can be translated to Dactl quite easily as follows:

$$P[x\ toc1\ toc2] \Rightarrow \#\#AND[\hat{}b1\hat{}\ b2], \quad b1 : *Unify[toc1\ Hello[x]], b2 : *Unify[toc2\ Hello[x]];$$

$$C[id\ Hello[x1 : Var]\ ANY] \Rightarrow *SUCCEED, x1 := *id \mid$$
$$C[id\ ANY\ Hello[x2 : Var]] \Rightarrow *SUCCEED, x2 := *id;$$
$$C[p1\ p2 : Var\ p3 : Var] \Rightarrow \#C[p1\ \hat{}p2\ \hat{}p3];$$
$$C[ANY\ ANY\ ANY] \Rightarrow *FAIL;$$

So assuming the following query:

$$: -p(x1, m11, m12), \quad p(x2, m21, m22), \quad c(a, m11, m21), \quad c(b, m12, m22).$$

which is translated to Dactl as:

$$\begin{aligned} INITIAL \Rightarrow & \#AND[\hat{}b1\ \hat{}b2\ \hat{}b3\ \hat{}b4], \\ & b1 : *P[x1 : Var\ m11 : Var\ m12 : Var], \\ & b2 : *P[x2 : Var\ m21 : Var\ m22 : Var], \\ & b3 : *C[A\ m11\ m21], \\ & b4 : *C[B\ m12\ m22]; \end{aligned}$$

when the process network terminates, one of $x1$ or $x2$ will be bound to A and the other to B. Incidentally, note the use of the predefined node *INITIAL* in the first rule; any Dactl computation commences with a graph comprising the single node *INITIAL*.

Finally, we consider the mappings of FCP(?) and FCP(:,?) to Dactl. We note that there are basically two uses for read-only variables. The first is in the implementation of test-and-set operations as shown by the following FCP(?) program:

$$test\_and\_set(X?, T) : -X = T.$$

where the intention is that X is unified only if it is a variable. These cases, however, can be taken care of in Dactl trivially, without the need for read-only annotations, using the techniques discussed so far:

$$Test\_and\_set[x : Var\ t] : - * SUCCEED, x := *t;$$

Incidentally, a similar technique can be used to implement Saraswat's **wait** annotation using the pattern $x : (ANY - Var)$.

The second use is in the creation of protected data structures. Consider the following FCP(:,?) program:

$$p(Xs, \dots) : - \dots true : Xs = [message \mid Xs'?] \mid p(Xs', \dots).$$

where the producer p protects the incomplete part of the stream it produces from the outside world. A possible implementation in Dactl is the following:

$$P[xs : Var \dots] \Rightarrow *P[xs' : Var \dots], xs := *Cons[Message\ RO[xs']];$$

where any other process will be accessing that incarnation of xs' protected with RO. A unification primitive implemented in Dactl should now be extended with rules handling the data structure RO:

$$Unify[v1 : Var\ v2 : RO[Var]] \Rightarrow *SUCCEED, v1 := v2;$$
$$Unify[RO[ro : RO[ANY]]\ term] \Rightarrow *Unify[ro\ term];$$
$$Unify[RO[v1 : Var]\ RO[v2 : Var]] \Rightarrow \#\#Unify[\hat{}v1\ \hat{}v2];$$

The second rule dereferences the nested *RO* structures produced in the unifications and the third rule chooses to suspend upon encountering two read only variables involved in a unification.

In general, the use of read-only annotations complicates significantly our model (pattern matching in the left-hand side of Dactl rules becomes also more complex) and we would like to avoid their use as much as possible. It should be emphasized that, in general, an embedding of FCP(?) in Dactl is highly problematic since it relies heavily on implicit head unification which is not supported by the weaker (in this respect) Dactl model.

4 Enhancing concurrent logic programs with functional capabilities

In this section we discuss the extension of the concurrent logic model with some features found usually only in functional languages such as lazy evaluation, sharing of computation, and higher order functions. The introduction of sharing enhances the efficiency of concurrent logic programs. Lazy evaluation is also very useful, but the enhancement of the 'eager' concurrent logic component with laziness introduces deadlock and we will discuss techniques for avoiding this.

4.1 GHC/F

We start with a brief description of a prototype language called GHC/F, a superset of GHC, which is used as a vehicle for the development of the above ideas. A GHC/F to Dactl compiler has been built as an extension of that for GHC described previously.

Programs in GHC/F consist of clauses and rewrite rules written in any of the following three forms:

$LHS :- G \mid B.$ (A)
$LHS \Rightarrow RHS.$ (B)
$LHS \Rightarrow RHS :- G \mid B.$ (C)

LHS is a relational atom of the form $p(t1,\ldots,tm)$ (type A) or a functional atom of the form $f(t1,\ldots,tm)$ (types B and C) where p is the name of the predicate, f is the name of the function and $t1$ to tm are **data terms**. A data term is either a variable, a constant, or a constructor of the form $d(t1,\ldots,tn)$ where d is the data constructor's name and $t1$ to tn are data terms. RHS can be either a data term or a functional atom of the form $f(t1,\ldots,tm)$ where $t1$ to tm are either data terms or functional atoms. The part $:- G \mid B$ is called the **condition**, where G and B are defined as in GHC. Note that the second form is identical to

$$LHS \Rightarrow RHS :- true \mid true.$$

Note also that some of a function's parameters may be output arguments. If that is the case however, these arguments must always be a variable; an attempt to instantiate an output argument which is not a variable will cause a run time exception whose result is undefined. In addition, a mode declaration identical to those used in Parlog must accompany the definition of the function distinguishing explicitly the input arguments from the output ones. In the absence of mode declarations, all the arguments of a function are treated as input by default.

GHC/F supports four unification primitives which can be used anywhere in the guard and/or body of a clause (A) or rewrite rule (C):

$=$ denotes lazy evaluation; $x = f(\ldots)$ will unify x with $f(\ldots)$ without evaluating f (this allows sharing of computations, as we will see below).

$:=$ is the strict assignment primitive; $x := f(\ldots)$ will assign x to $f(\ldots)$ and at the same time fire f; if at the time of the call x is not a variable, an error will be reported.

There are also the strict unification primitive $=:=$ and the one-way unification primitive $\Leftarrow$; these are discussed elsewhere (Glauert and Papadopoulos 1991). Note also that GHC/F supports the operator @ used in higher-order programming. In particular, $f@(X)$ where X represents an argument or a list of arguments will produce the function or predicate application $f(X)$ depending on whether the metavariable f represents a function or predicate. The use and implementation in Dactl of this primitive is described in Papadopoulos (1989a).

We illustrate GHC/F's expressive power with three examples; a complete description of the language is given in Glauert and Papadopoulos (1991). The first example computes efficiently the Fibonacci numbers using extra **output variables** for storing intermediate results (Josephs 1986):

$mode\ fib(?,\hat{},\hat{}).$

$fib(0,X,\_) \Rightarrow 1 :- true \mid X := 1.$
$fib(1,X,Y) \Rightarrow 1 :- true \mid X := 1, Y := 1.$
$fib(N,X,Y) \Rightarrow fib(N1,Y,Z) :- true \mid N1 := N-1, X := Y+Z.$

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*. Note also that it is the user's responsibility

to ensure that such output variables have only one producer function (i.e. they do not appear as output arguments of more than one function). In fact, the above program is more efficient than the fully lazy version given by Josephs (1986), since more needed parallelism can be extracted.

The second example is an implementation of the quicksort algorithm:

$$sort(List, Sorted) : -true \mid qsort(List, Sorted, []).$$

$$\begin{aligned} qsort([U \mid X], Sorted\_h, Sorted\_t) : -true \mid\ & X1 := partition(U, X, X2), \\ & qsort(X1, Sorted\_h, [U \mid Sorted]), \\ & qsort(X2, Sorted, Sorted\_t). \end{aligned}$$

$$qsort([], Sorted\_h, Sorted\_t) : -true \mid Sorted\_h = Sorted\_t.$$

$$mode\ partition(?, ?, \hat{}).$$

$$partition(U, [V \mid X], X2) \Rightarrow [V \mid X1] : -U < V \mid X1 := partition(U, X, X2).$$

$$partition(U, [V \mid X], X2) \Rightarrow partition(U, X, X2') : -V =< U \mid X2 := [V \mid X2'].$$

$$partition(\_, [\,], X2) \Rightarrow [\,] : -true \mid X2 := [\,].$$

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 last example is a Hamming numbers generator and we illustrate here the use of sharing and the handling of infinite data structures.

$$\begin{aligned} hamming() \Rightarrow H : -true \mid\ & H = [1 \mid merge(X2, merge(X3, X5))], \\ & X2 := times(2, H), \\ & X3 := times(3, H), X5 := times(5, H). \end{aligned}$$

$$merge([U \mid X], [V \mid Y]) \Rightarrow [U \mid merge(X, Y)] \quad : -U == V \mid true.$$

$$merge([U \mid X], [V \mid Y]) \Rightarrow [U \mid merge(X, [V \mid Y])] \quad : -U < V \mid true.$$

$$merge([U \mid X], [V \mid Y]) \Rightarrow [V \mid merge([U \mid X], Y)] \quad : -V < U \mid true.$$

$$times(U, [V \mid Y]) \Rightarrow [W \mid times(U, Y)] \quad : -true \mid mul(U, V, W).$$

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 eagerly, 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.

4.2 Mapping GHC/F programs to Dactl: the deadlock problem

The principles of translating GHC/F programs to equivalent sets of Dactl rewrite rules have been discussed elsewhere (Papadopoulos 1989a). They rely on integrating the techniques described in the previous sections with those developed for implementing functional languages in Dactl and in particular ML (Hammond 1990) and Clean (Kennaway 1988). However, the interaction of the 'eager' concurrent logic part with the ('lazy' subset of the) functional one introduces deadlock, particularly in the presence of

shared variables. The problem of deadlock occurs in many logic+functional computational models and there are numerous methods for detecting it and resolving it. In residuation (Hanus 1992), for instance, deadlock is detected by compile time analysis which, however, yields only an approximation of the actual operational behaviour of a program. In GHC/F deadlock is detected and resolved at run time. This has the additional advantage of retaining the compositionality potential of the original language (GHC); any GHC/F program component can interact with other independent components and producer–consumer relationships in the presence of logic variables in lazy functions will be determined at run time.

Deadlock problems in GHC/F arise in two cases, both having to do with instantiation of variables to data terms or function applications. The first case involves the use of the lazy unification primitive: if, in addition to a call $x = f(\dots)$ where $f(\dots)$ is a function application, some other process is suspended on x waiting for its value, then we should evaluate $f(\dots)$ even if $=$ is lazy. This, however, may not be easy to detect, as illustrated by the following example:

$$?\dots prod(X), cons(X).$$
$$prod(X) : -true \mid X = f(\dots).$$
$$cons(1).$$

where $f(\dots)$ is a function application. Using the techniques described in Glauert and Papadopoulos (1991), the above program would be translated to Dactl as follows (in this section we ignore for simplicity the rules needed for the run time test):

$$INITIAL \Rightarrow *Prod[x : Var], *Cons[x];$$
$$Prod[x] \Rightarrow *LazyUnify[x\ F[\dots]];$$
$$Cons[1] \Rightarrow *SUCCEED;$$
$$Cons[x : Var] \Rightarrow \#Cons[\hat{}x];$$
$$Cons[f : REWRITABLE] \Rightarrow \#Cons[\hat{}*f];$$

Note that the third rule of *Cons* will fire its argument if it matches the pattern *REWRITABLE*, i.e. if it is a function. All node identifiers representing functions are defined to match the generic pattern *REWRITABLE*. Note also that *LazyUnify* does not fire $F[\dots]$; in particular, *LazyUnify* is defined by rules of the form:

$$LazyUnify[v : Var\ vr : (Var + REWRITABLE)] \Rightarrow *SUCCEED, v := vr;$$
$$LazyUnify[v : Var\ data\_term] \Rightarrow *SUCCEED,\ v := *data\_term;$$

Now, if *Prod* completes execution before *Cons* is tried for reduction, then the common variable x will have already been instantiated to $F[\dots]$ and *Cons* will be able to fire it (by matching the third rule) and get the required result. If, however, *Cons* is executed first, then it will suspend on the variable x until the latter is instantiated. The problem here is that *Prod* cannot possibly know whether anyone (*Cons* in this case) is waiting for x's value, and the lazy unification primitive will assign $F[\dots]$ to x without firing it, thus causing deadlock.

To solve the problem, we pass the responsibility of indicating that the value of a variable is needed by some process to the process itself. Any waiting process must now change the pattern of any input variable

from *Var* to *NVar* (say). The lazy unification primitive then can recognize that the value for *NVar* must be computed and fire the appropriate closure. Using this technique the procedure *Cons* must be compiled to *Dactl* as follows:

$$Cons[1] \Rightarrow *SUCCEED;$$
$$Cons[x : Var] \Rightarrow \#Cons[\hat{}x], x := NVar;$$
$$Cons[x : NVar] \Rightarrow \#Cons[\hat{}x];$$
$$Cons[f : REWRITABLE] \Rightarrow \#Cons[\hat{}*f];$$

A Dactl implementation of the lazy unification primitive $=$ must now include the following rules:

$$LazyUnify[nv : NVar\ t : (ANY - Var)] \Rightarrow *SUCCEED, nv := *t;$$
$$LazyUnify[v : Var\ t : (Var + NVar + REWRITABLE)] \Rightarrow *SUCCEED, v := t;$$
$$LazyUnify[v : Var\ data\_term] \Rightarrow *SUCCEED, v := *t;$$

The second case causing deadlock is when a function making use of output variables (as in *fib* and *partition* earlier on) interacts with a predicate treating these variables as input arguments. If the function is lazy there will be deadlock since the predicate will be suspended waiting for the variables' bindings while at the same time the lazy function will need someone to trigger its evaluation. The problem can be alleviated by restricting the use of output variables to eager functions (Josephs 1986). A possible solution, however, is to associate these sort of variables with their defining functions; if a predicate encounters during input unification such a variable in a head's non-variable input argument, it activates the corresponding function. This case is reminiscent to the one where we associate variables with environments in the implementation of GHC's run time test; there is also some similarity with K-LEAF's *prodvar* term (Bosco et al. 1989). Consider the following base case:

$$mode\ f(\hat{}).$$
$$prod(X) : -true \mid X = f(Y), cons(Y).$$
$$cons(1).$$

where Y is an output variable of some function f. Translation to Dactl using the techniques described so far gives:

$$Prod[x\ y] \Rightarrow *LazyUnify[x\ F[y : Var]], *Cons[y];$$

Cons is defined as above.

Note that *LazyUnify* will not fire F and, unless somebody demands the value of x, *Cons* will remain suspended for ever on the variable y. The problem can be solved by associating y with its defining function F as follows:

$$Prod[x\ y] \Rightarrow *LazyUnify[x\ f : F[y : Var[f]]], *Cons[y];$$

where f is the identifier of the graph node representing the function F. The implementation of *Cons* now includes the following rule:

$$Cons[v : Var[f]] \Rightarrow \#Cons[\hat{}v], *f;$$

Table 4: Performance comparison: hamming and quicksort

Programs	Hamming				Quicksort			
Langs	**R**	**PC**	**AvP**	**MxP**	**R**	**PC**	**AvP**	**MxP**
Clean	672	1018	1.14	4	6654	11 357	1.11	2
GHC	3471	1098	4.98	30	16 879	944	28.85	60
GHC/F	1202	736	2.60	9	10 255	844	19.77	45

R, rewrites; PC, parallel cycles performed; AvP, activations processed per cycle (mean value); MxP: activations processed per cycle (peak value).

Since the output variables of a function are now pointing to that function, any computation that needs their value can activate the function through them. Again, here, the very fact that Dactl has no true variables has once more come to our rescue. As in the case of GHC's run time test, a variable can be viewed as being an intelligent object associated with useful information regarding its identity, surrounding defining environment, etc.

In the presence of output variables in functions as described above, a number of issues arise related to their interaction with the rest of the computation. These are fully discussed in Papadopoulos (1989a), but to give an idea of what is involved we note that in a unification $X = Y$ where both X and Y are output variables of some functions, the lazy unification primitive $=$ will fire these functions to determine whether the produced values are compatible:

$$LazyUnify[v1 : Var[f1 : REWRITABLE]v2 : Var[f2 : REWRITABLE]] \Rightarrow \#\#LazyUnify[\hat{}v1\ \hat{}v2], *f1, *f2;$$

This behaviour stems from the fact that an output variable of a function is not a pointer to some empty slot in the abstract memory space of the computation (as is the case for an 'ordinary' variable) but rather a pointer to a specific computation. There is also the need to restrict the transitions between different kinds of variables to valid ones: a valid transition is one that does not cause loss of information. For instance, the redirection $v1 : Var := v2 : Var[r]$ is legal but the opposite is illegal since the information r (whether a pointer to a function or a local environment) would be lost. All the valid transitions between different kinds of variables are given in Papadopoulos (1989a).

As before, we end the section with some performance analysis (see Table 4). We compare three versions of hamming and quicksort written in Clean, GHC and GHC/F. Both the Clean to Dactl programs are lazy. The GHC to Dactl programs have been translated using the compiler described in section 1.3; lazy evaluation is achieved in GHC by using the technique of changing the producer into 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 (as before) sorts a reversed list of 50 elements.

The results are quite encouraging since 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.

5 Conclusions, related and further work

In Saraswat et al. (1988) it is argued that certain computational behaviours are more tractable in one programming language framework than another, allowing for easier understanding and reasoning. In

this paper we have shown that a language based on graph rewriting can provide such a framework for declarative computational models. In particular, we have presented a methodology for mapping a wide class of concurrent logic languages onto a graph rewriting computational model using the intermediate language Dactl as a vehicle. We showed how the features of these languages such as unification (ask and tell), synchronization mechanisms, etc. can be modelled in a graph rewriting model by taking advantage of the latter's features such as node sharing and multiple redirections. Compilers have been written that translate concurrent logic programs to equivalent sets of Dactl rewrite rules. These can take advantage of techniques developed for the efficient execution of Dactl programs such as native machine code generation (King and Glauert 1991). We have also illustrated the potential of graph rewriting in amalgamating logic and functional features within a unified framework.

Our work, combined with other similar attempts (Darlington et al. 1988, Kennaway 1988, Hammond 1990), shows that graph rewriting can be seen as a general framework encompassing a variety of computational models and in particular logic and functional ones. The model has also been used for other purposes such as modelling process calculi (Glauert 1992). A promising area of research would be to study its potential for supporting object oriented programming along the lines proposed in Dami (1992) but also with respect to concurrent OOP based on the actor model. Further, the possibility of interacting objects with predicates and functions within the unified graph rewriting framework should also be explored.

Recently it has been shown that concurrent logic languages are instances of the wider family of constraint logic programming languages (Saraswat 1989). Moreover, Montanari and Rossi (1991) discuss the relationship of concurrent constraint logic programming and graph rewriting. A fruitful area of research is towards extending the work presented in this paper to provide graph rewriting based implementations of concurrent constraint logic languages.

In addition, parallel logic programming and concurrent logic programming have been combined into a hybrid model known as the Andorra family of languages combining stream AND-parallelism with OR-parallelism. Languages that belong to this category include, among others, Andorra-I (Santos Costa et al. 1991), AKL (Janson and Haridi 1991) and Pandora (Bahgat 1991). Although it is not shown in this paper, we can implement these languages in Dactl using high-level transformation techniques like the ones discussed in Bahgat (1991) where Pandora programs are translated to Parlog equivalent ones.

Furthermore, we believe that graph rewriting has a role to play in the development of open systems and the language support that this will require. There is a trend here towards multilingual paradigms such as compositional notations allowing the writing of programs composed of subprograms written in different languages (logic, functional or even imperative) and their interaction in a parallel environment (Chandry and Kesselman 1992). Although languages like Strand or Janus discussed in this paper can play the role of a unifying operational model comprising multilingual parallel processes, we believe that more general computational models such as graph rewriting systems are more suitable for this purpose. To this end, we have done some initial work (Banach and Papadopoulos 1993) where we show that most of the languages discussed in this paper can be translated without loss of expressiveness to a weaker subset of Dactl (Banach 1993) especially suited for distributed environments.

Acknowledgements

Most of this work was done while the author was a member of the Declarative Systems Project (DSP) team at the University of East Anglia. Enlightening discussions with many people, there and elsewhere, including Kevin Hammond, Richard Kennaway, Ronan Sleep, Richard Banach and Nic Holt have helped

enormously. John Glauert deserves special thanks for being a constant source of inspiration.

References

Aït-Kaci, H. (1991) *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, Cambridge, MA.

Bahgat, R. (1991) Pandora: non-deterministic parallel logic programming. Ph.D. Thesis, Imperial College, London, UK.

Banach, R. (1993) MONSTR: term graph rewriting for parallel machines, *in Term Graph Rewriting: Theory and Practice* (eds. M.R. Sleep, M.J. Plasmeijer and M.C.J.D. van Eekelen), Wiley, New York, pp. 243–52.

Banach, R. and Papadopoulos, G.A. (1993) Parallel term graph rewriting and concurrent logic programs, *in WP&DP93*, Sofia, Bulgaria, 4–7 May, North-Holland, Amsterdam, pp. 303–22.

Banach, R. and Watson, P. (1988) Dealing with state on Flagship: the Monstr computational model, *in CONPAR88*, Manchester, UK, 12–16 September, Cambridge University Press, Cambridge, pp. 595–604.

Barendregt,H.P., Eekelen, M.C.J.D., Glauert, J.R.W.J., Kennaway, R., Plasmeijer, M.J. and Sleep, M.R. (1987) Term graph rewriting, *in PARLE87*, Eindhoven, The Netherlands, 15–19 June, Lecture Notes in Computer Science **259**, Springer-Verlag, Berlin, pp. 141–58.

Bosco, P.G., Cecchi, C., and Moiso, C. (1989) IDEAL and K-LEAF implementation: a progress report, *in PARLE89*, Eindhoven, The Netherlands, 12–16 June, Lecture Notes in Computer Science **365**, Springer-Verlag, Berlin, pp. 413–32.

Dami, L. (1992) Hierarchical objects with ports, *in Object Frameworks* (ed. D. Tsichritzis), Centre Universitaire d' Informatique, Université de Genève, pp. 41–78.

Darlington, J.and Reeve, M. (1981) Alice – a multiprocessor reduction machine for the parallel evaluation of applicative languages, *in ACM FPLCA81*, New Hampshire, 1981, pp. 65–75.

Darlington, J., Khoshnevisan, H., McLoughlin, L.M.J., Perry, N., Pull, H.M., Sephton, K.M. and While, R.L. (1988) An introduction to the Flagship programming environment, *in CONPAR88*, Manchester, UK, 12–16 September, Cambridge University Press, Cambridge, pp. 108–15.

Chandry, K.M. and Kesselman, C. (1992) The derivation of compositional Programs, *in JICSLP92*, Washington, DC, USA, 9–14 November, MIT Press, pp. 3–17.

Dershowitz, N. (1985) Computing with rewrite rules. *Information and Control* **65**, 122–57.

Furukawa, K. (1991) Fifth generation computer project: towards large-scale knowledge information processing, *in ISLP91*, San Diego, USA, 28 October–1 November, MIT Press, Cambridge, MA, pp. 719–31.

Foster, I. and Taylor, S. (1990) *Strand: New Concepts in Parallel Programming.* Prentice Hall, Englewood Cliffs, NJ.

Foster, I. and Winsborough, W. (1991) Copy avoidance through compile time analysis and local reuse, *in ISLP91*, San Diego, USA, 28 October–1 November, MIT Press, Cambridge, MA, pp. 455–69.

Glauert, J.R.W. (1992) Asynchronous mobile processes and graph rewriting, *in PARLE92*, Paris, France, 15–18 June, Lecture Notes in Computer Science, Springer-Verlag, Berlin, pp. 63–78.

Glauert, J.R.W. and Papadopoulos, G.A. (1988) A parallel implementation of GHC, *FGCS88*, Tokyo, Japan, 28 November–2 December, Vol. **3**, pp. 1051–8.

Glauert, J.R.W. and Papadopoulos, G.A. (1991) Unifying concurrent logic and functional languages in a graph rewriting framework, *in 3rd Panhellenic Computer Science Conference*, Athens, Greece, 26–31 May, Vol. **1**, pp. 59–68.

Glauert, J.R.W., Kennaway, J.R., Sleep, M.R. and Somner, G.W. (1988a) Final Specification of Dactl. Internal Report SYS-C88-11, University of East Anglia, UK.

Glauert, J.R.W., Hammond, K., Kennaway, J.R. and Papadopoulos, G.A. (1988b) Using Dactl to implement declarative languages, *in CONPAR88*, Manchester, UK, 12–16 September, Cambridge University Press, pp. 116–24.

Glauert, J.R.W. Kennaway, J.R., Papadopoulos, G.A. and Sleep, M.R. (1997) Dactl: an experimental graph rewriting language. *Journal of Programming Languages*, **5**, 75–98.

Gregory, S. (1987) *Parallel Logic Programming in Parlog – The Language and its Implementation*. Addison-Wesley, London.

Gregory, S., Foster, I., Burt, A.D. and Ringwood, G.A. (1989) An abstract machine for the implementation of Parlog on uniprocessors. *New Generation Computing*, **6**(4), 389–420.

Hammond, K. (1990) *Parallel SML: A Functional Language and its Implementation in Dactl.* Pitman, London.

Hammond, K. and Papadopoulos, G. A. (1988) Parallel implementation of declarative languages based on graph rewriting, *in UK IT'88*, Swansea, UK, 4–7 July, pp. 246–9.

Hanus, M. (1992) On the completeness of residuation, *in JICSLP92*, Washington, DC, USA, 9–14 November, MIT Press, Cambridge, MA, pp. 192–206.

Janson, S. and Haridi, S. (1991) Programming paradigms of the Andorra kernel language, *in ISLP91*, San Diego, USA, 28 October–1 November, MIT Press, Cambridge, MA, pp. 167–86.

Josephs, M. B. (1986) Functional programming with side effects. *Science of Computer Programming*, **7**, 279–96.

Kennaway, J.R. (1988) Implementing term rewrite languages in Dactl, *in CAAP88*, Nancy, France, 21–24 March, Lecture Notes in Computer Science **299**, Springer-Verlag, Berlin, pp. 117–31.

King, I. and Glauert, J.R.W. (1991) Generating native code for a generalised graph rewriting language, *ESPRIT BRA 3074 Deliverable A1.1*.

Levy J. and Shapiro, E.Y. (1987) Translation of safe GHC and safe concurrent Prolog to FCP, *in Concurrent Prolog Collected Papers* (ed. E.Y. Shapiro), MIT Press, London, Vol. **2**, pp. 384–414.

Montanari, U. and Rossi, F. (1991) True concurrency in concurrent constraint programming, *in ISLP91*, San Diego, USA, 28 October–1 November, MIT Press, Cambridge, MA, pp. 694–713.

Nilsson, M. and Tanaka, H. (1986) FLENG Prolog – the language which turns supercomputers into parallel machines, *Logic Programming '86*, Tokyo, Japan, 23–26 June, Lecture Notes in Computer Science **264**, Springer-Verlag, Berlin, pp. 170–9.

Papadopoulos,G.A. (1989a) Parallel implementation of concurrent logic languages using graph rewriting techniques. Ph.D. Thesis, University of East Anglia, UK.

Papadopoulos, G.A. (1989b) A fine grain parallel implementation of Parlog, *in TAPSOFT89*, Barcelona, Spain, 13–17 March, Lecture Notes in Computer Science **352**, Springer-Verlag, Berlin, pp. 313–27.

Peyton Jones, S.L., Clack, C., Salkild, J. and Hardie, M. (1987) GRIP – a high performance architecture for parallel graph reduction, *in FPLCA87*, Portland, Oregon, USA, 14–16 September, Lecture Notes in Computer Science **274**, Springer-Verlag, Berlin, pp. 98–112.

Procter, B.J. and Skelton, C.J. (1988) Flagship is nearing port, *in CONPAR88*, Manchester, UK, 12–16 September, Cambridge University Press, Cambridge, pp. 100–7.

Santos Costa, V., Warren, D.H.D. and Yang, R. (1991) The Andorra-I engine: a parallel implementation of the basic Andorra model, *in ICLP91*, Paris, France, 24–28 June, MIT Press, Cambridge, MA, pp. 825–39.

Saraswat, V.A. (1989) Concurrent constraint programming languages. Ph.D. Thesis, Carnegie-Mellon University (published by ACM Doctoral Dissertation Award series, MIT Press, 1993).

Saraswat, V.A., Weinbaum, D., Kahn, K. and Shapiro, E. (1988) Detecting stable properties of networks in concurrent logic programming languages, *ACM PODC88*, pp. 210–22.

Saraswat, V.A., Kahn, K. and Levy, J. (1990) Janus: a step towards distributed constraint programming, *in NACLP90*, Austin, USA, 29 October–1 November, MIT Press, Cambridge, MA, pp. 431–46.

Shapiro, E.Y. (1989) The family of concurrent logic programming languages. *Computing Surveys*, **21** (3), 412–510.

Silverman, W., Hirsch, M., and Houri, A. (1988) *The Logix System User Manual Ver. 2.0*. Technical Report CS-21, Weizmann Institute, Israel.

Steel, T.B. (1961) UNCOL: the myth and the fact. *Annual Review in Automated Programming* **2**, 325–44.

Tick, E. (1991) *Parallel Logic Programming*. MIT Press, London.

Ueda, K. (1986) Guarded Horn clauses. D.Eng. Thesis, University of Tokyo, Japan.

Warren, D.H.D. (1983) *An Abstract Prolog Instruction Set*. SRI International, CA, USA.

Watson, I., Sergeant, J., Watson, P. and Woods, V. (1988) The flagship parallel machine, *in CONPAR88*, Manchester, UK, 12–16 September, Cambridge University Press, Cambridge, pp. 125–33.