ELSEVIER

Microprocessors and Microsystems 20 (1997) 511–520

MICROPROCESSORS AND
MICROSYSTEMS

Experience using an intermediate compiler target language for parallel machines

George A. Papadopoulos*

Department of Computer Science, University of Cyprus, 75 Kallipoleos Str., P.O. Box 537, CY-1678, Nicosia, Cyprus

Received 4 September 1996; revised 18 November 1996; accepted 25 November 1996

Abstract

The generalised computational model of term graph rewriting systems (TGRSs) has been used extensively as an implementation vehicle for a number of, often divergent, programming paradigms ranging from the traditional functional programming ones to the (concurrent) logic programming ones and various amalgamations of them, to (concurrent) object-oriented ones. More recently, the relationship between TGRSs and process calculi (such as the π-calculus) as well as linear logic has also been explored. In this paper we describe our experience in using the intermediate compiler target language Dactl based on TGRSs for mapping a variety of programming paradigms of the aforementioned types onto it. In particular, we concentrate on some of the issues that we feel have played an important role in our work (in, say, affecting performance, etc.), the aim being to derive a list of features that we feel every language model which intends to be used as an intermediate representation between (concurrent) high-level languages and (parallel) computer architectures must have. © 1997 Elsevier Science B.V.

Keywords: Compiler target languages; Intermediate representations; Parallel and distributed computing; Language embeddings; Term graph rewriting systems

1. Introduction

The concept of deriving a generalised computational model able to act as an interface between a variety of high-level languages and computer architectures is as old as (postwar) computer science itself and it is best envisaged by Steel's paper on the mythical Universal Computer Language (UNCOL) [40]; see Fig. 1.

There are a number of advantages in having such an intermediate computational model, namely: (i) reduction of the number of implementations for M languages and N machine architectures from $M \times N$ to $M + N$; (ii) decoupling of language development from architecture development; (iii) exploitation of the intermediate formalism to act as a two-way interface between different languages and architectures, thus effectively offering multilinguality and (loose) multistyle machine integration; (iv) from the software engineering point of view, different research and development groups can concentrate at their level of expertise (language design, architecture design, compilation techniques, etc.) without the need to be aware of and familiar with the technical details of the work done at other levels not directly related to theirs.

Furthermore, from the language design point of view, there are some more specific advantages, namely: (i) easy and fast modification of prototype implementations of some language onto a (high-level) intermediate formalism and ability to effectively assess the usefulness and correct design of new language features or extensions; (ii) use of the intermediate formalism as a common basis for comparing different language models but also as a means of enhancing one language with features of others (amalgamation of languages).

The introduction of such an intermediate level of computation is not, of course, without its problems and dilemmas, some of them being some overhead incurred from the additional level of code mapping, whether the intermediate formalism should be closer to the languages' level or the machine level, whether it should constitute a concrete language or a set of 'add on' primitives, and, finally, how to resolve successfully the issue of temporal vs. spatial segregation [29].

In designing such an intermediate formalism some of the things that must be considered are the following: (i) information necessary to generate efficient code for a wide variety of machines must be represented in or be accessible to the intermediate formalism while the latter should exploit this information to generate 'reasonably efficient' code

* E-mail: george@turing.cs.ucy.ac.cy

PII S0141-9331(96)01123-4

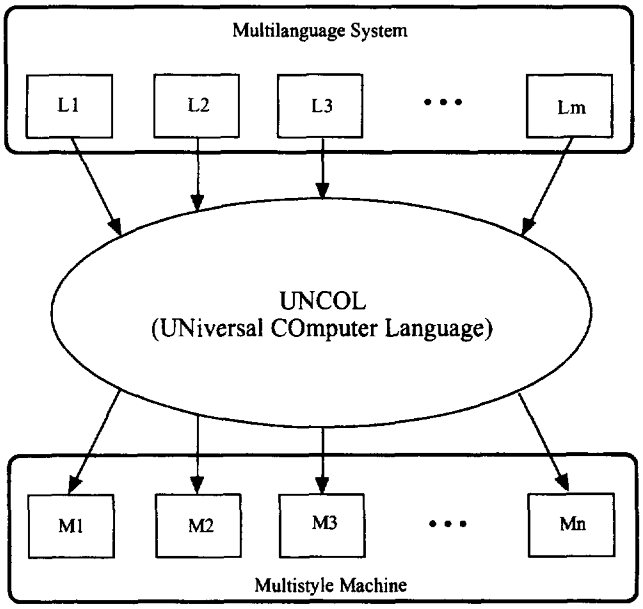


Fig. 1. UNCOL.

(meaning that the benefits gained from using the intermediate formalism outweigh the overhead incurred in the extra implementation level introduced); (ii) the intermediate formalism should be based on some abstract model of computation which is expressive enough to accommodate the needs of developing software.

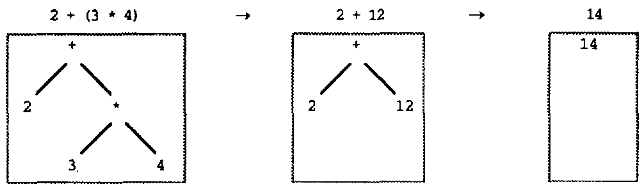
From this rather general discussion on intermediate formalisms, it becomes immediately apparent that if a particular formalism is to be successful in playing the role of UNCOL, it must concentrate on specific language and architecture families. Some well-known intermediate formalisms and associated abstract computational models that have been proposed are Linda's tuple space [1], Compositional Programming [10], the Program Composition Notation [14] and the model by Bisiani and Forin [8] but, in a way, also the λ-calculus, the SKI (and other) combinators and the highly successful Warren Abstract Machine (WAM) [42]. A collection of such intermediate formalisms (which is nowadays rather out of date) can be found in [11] and [32].

In this paper we will describe our experiences in using such an intermediate formalism and we will derive a set of features which we feel are essential and must be supported by this formalism. The computational model in question which acts as an interface between languages and machine architectures is that of term graph rewriting systems [6] and, more to the point, its associated compiler target language Dactl [17-19]. The rest of the paper is organised as follows: the next section introduces briefly TGRSs, the language Dactl and its role as an intermediate formalism and the following one discusses those features of the language that we judge to have played an important role in the success of Dactl as an intermediate compiler target language; the paper ends with some conclusions and discussion on related work.

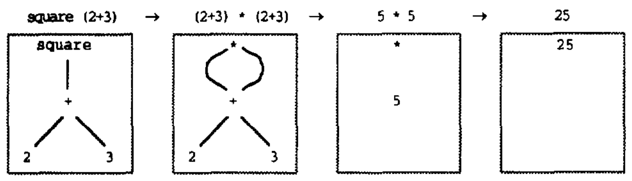
2. TGRSs and the compiler target language Dactl

Term graph rewriting systems [6,39], and in general rewriting systems [22], 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. It follows that a language based on rewriting theory has the potential of being an intermediate language for a number of declarative languages. A number of languages based on TGRSs have been designed and implemented but here we will concentrate on one of them, namely Dactl [17-19], which we believe is the most flexible of all. Dactl was essentially the target language of the Flagship project [27], part of the Alvey [37] and EDS [13] programmes, the purpose of which was the design and implementation of a parallel computing system able to support different declarative (logic, functional and object-oriented) programming paradigms.

As an illustrative example we can view the evaluation of the following expression as rewriting of terms or trees, which are restricted forms of graphs:



The usefulness of graph representation is revealed if we have a call to a function square which is evaluated lazily:



In the Dactl representation of these computations, even primitive arithmetic operations are represented as functions. A possible encoding of the above functions is the following:

```
MODULE ExprEx;
IMPORTS Arithmetic;
RULE
INITIAL ⇒ #IAdd[ 2 *IMul[ 3 4 ] ];
ENDMODULE ExprEx;

MODULE SquareEx;
IMPORTS Arithmetic;
SYMBOL REWRITABLE Square;
RULE
Square[ n ] ⇒ #IMul[ *n n ];
INITIAL ⇒ *Square[ IAdd[ 2 3 ] ];
ENDMODULE SquareEx;
```

The symbols *, # and are used to control the order of evaluation as explained in detail below. For now it suffices to say that the symbol * is a *spark* indicating the next expression to be reduced. It is necessary to make the control of evaluation explicit in Dactl since the notation will be used to model computations requiring a range of different evaluation strategies.

The sequence of evaluation in these cases is as follows (where `INITIAL` denotes the first rule in a Dactl program to be rewritten):

```
*INITIAL → #IAdd[ 2 *IMul[ 3 4 ] ] →
 #IAdd[ 2 *12 ] → *IAdd[ 2 12 ] → *14

*INITIAL → *Square[ IAdd[ 2 3 ] ] →
 #IMul[ *n:IAdd[ 2 3 ] n ] → #IMul[ *n:5 n ] →
 *IMul[ n:5 n ] → *25
```

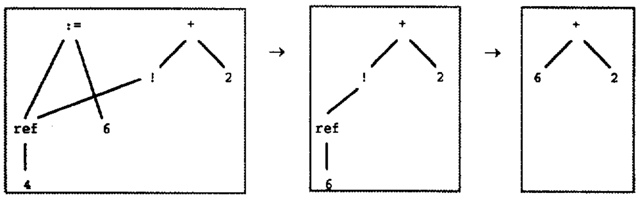
Note the use of the identifier `n` to indicate the sharing of the sub-expression `IAdd[2 3]`.

In Dactl it is possible to program with state where variables involved in a computation can have mutable values. The language SML, for instance, uses the concept of reference values which capture the concept in a way which can be modelled by graph rewriting. A function `ref` creates reference values with an initial value; the operator := is a function which assigns a new value to a reference and returns the unit value. The operator ! is used for dereferencing.

We model reference values as nodes whose contents can change during evaluation. Consider the expression:

```
let val r = ref (4)
in r: = 6; !r + 2
end
```

Sequencing of evaluation ensures that `r` holds the value 6 by the time the value is dereferenced:



The corresponding Dactl code for this program could be the following:

```
MODULE RefEx;
IMPORTS Arithmetic;
SYMBOL CREATABLE Unit;
SYMBOL OVERWRITABLE Ref;
SYMBOL REWRITABLE Assign; DeRef; Seq;
LetRes;
RULE
INITIAL ⇒ #Seq[ *Assign[r 6] LetRes[ r ] ], r: Ref[4];
LetRes[r] ⇒ #IAdd[ *DeRef[r] 2 ];
Assign[r:Ref[o] n] ⇒ *Unit, r: = Ref[n];
DeRef[Ref[v]] ⇒ *v;
Seq[ Unit b] ⇒ *b;
ENDMODULE RefEx;
```

The novel feature is in the rule for assignment. The `Assign` node is rewritten to a unit value, but at the same time, the first argument, a `Ref` node, is *overwritten* with a new `Ref` node with different contents. Other parts of the graph with pointers to this node will now find a different value if they apply the dereferencing operation to the node. `LetRes` represents the computation required after the first statement of the sequence has completed.

Variables in conventional languages correspond to the references of SML considered above. Variables in logic programming languages have a very different meaning, but similar graph rewriting techniques may be used.

Also, Dactl control markings are used to synchronise computation. If a goal can neither succeed nor fail until a variable has been instantiated, the computation suspends waiting for the variable to be given a value by output unification in some other goal. When this takes place, the original computation is reactivated and will be able to make further progress.

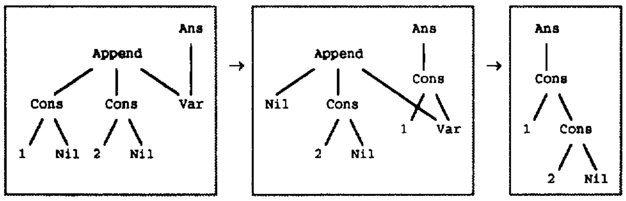
The example below, the unavoidable append program, serves to illustrate some of the techniques used in modelling a concurrent logic program [38] as a set of graph rewriting rules:

```
append([H|T],Y,Z) :- true | Z = [H|Z1],
append(T,Y,Z1).
append([], Y,Z) :- true | Z = Y.
?- append(P,Q,Ans), P = [1], Q = [2].
```

The corresponding Dactl code for this program could be the following:

```
MODULE Append;
SYMBOL CREATABLE Ans; Cons; Nil;
SYMBOL REWRITABLE Append;
SYMBOL OVERWRITABLE Var;
RULE
INITIAL ⇒ Ans[ans:Var], *Append[p q ans],
p:Cons[1 Nil], q:Cons[2 Nil].
Append[Cons[h t] y z:Var] → z: = *Cons[h zz:Var],
*Append[t y zz];
Append[Nil y z:Var] → z: = *y;
Append[l:Var y z] → #Append[l y z];
ENDMODULE Append;
```

There follows a graphical representation of the execution behaviour of the above program:



In general, a Dactl rule is of the form

Pattern → Contractum, x_1: = y_1,...,x_i: = y_i, $\mu_1 z_1 \ldots \mu_j z_j$

where after matching the `Pattern` of the rule with a piece of the graph representing the current state of the computation, the `Contractum` is used to add new pieces of graph to the existing one and the redirections $x_1 := y_1, \ldots, x_i := y_i$ are used to redirect a number of arcs (where the arc pointing to the root of the graph being matched is usually also involved) to point to other nodes (some of which will usually be part of the new ones introduced in the `Contractum`); the last part of the rule $\mu_1 z_1 \ldots \mu_j z_j$ specifies the state of some nodes (idle, active or suspended).

The `Contractum` is also a Dactl graph where however the definitions for node identifiers that appear in the `Pattern` need not be repeated. So, for example, the following rule

r:F[x:(ANY-INT) y:(CHAR + STRING) v1:REWRITABLE v2:REWRITABLE]
→ ans:True, d1:1, d2:2, r: = *ans, v1: = *d1, v2: = *d2;

will match that part of a graph which is rooted at a (rewritable) symbol `F` with four descendants where the first matches anything (ANY) but an integer, the second either a character or a string and the rest overwritable symbols. Upon selection, the rule will build in the contractum the new nodes `ans`, `d1` and `d2` with patterns `True`, `1` and `2` respectively; finally, the redirections part of the rule will redirect the root `F` to `ans` and the sub-root nodes `v1` and `v2` to `1` and `2` respectively. The last two non-root redirections model effectively assignment. A number of syntactic abbreviations can be applied which lead to the following shorter presentation of the above rule

F[x:(ANY-INT) y:(CHAR + STRING) v1:REWRITABLE v2:REWRITABLE] ⇒ *True, v1: = *1, v2: = *2;

where ⇒ is used for root overwriting and node identifiers are explicitly mentioned only when the need arises. Finally, note that all root or sub-root overwritings involved in a rule reduction are done atomically. So in the above rule the root rewriting of `F` and the sub-root rewritings of `v1` and `v2` will all be performed as an atomic action.

The way computation evolves is dictated not only by the patterns specified in a rule system but also by the control markings associated with the nodes and arcs of a graph. In particular, * denotes an active node which can be rewritten and $\#^n$ denotes a node waiting for n notifications. Notifications are sent along arcs bearing the notification marking. Computation then proceeds by arbitrarily selecting an active node `t` in the execution graph and attempting to find a rule that matches at `t`. If such a rule does not exist (as, for instance, in the case where `t` is a constructor) notification takes place: the active marking is removed from `t` and a 'notification' is sent up along each -marked in-arc of `t`. When this notification arrives at its (necessarily) $\#^n$-marked source node `p`, the mark is removed from the arc, and the `n` in the $\#^n$ marking of `p` is decremented. Eventually, $\#^0$ is replaced by *, so suspended nodes wake when all their subcomputations have notified.

Now suppose the rule indeed matches at active node `t`. Then the RHS of that rule specifies the new markings that will be added to the graph or any old ones that will be removed. In the example above, for instance, the new nodes `ans`, `d1` and `d2` are activated. Since no rules exist for their patterns (`True`, `1` and `2` are 'values'), when their reduction is attempted, it will cause the notification of any node bearing the # symbol and its immediate activation. This mechanism provides the basis for allowing a number of processes to be coordinated with each other during their, possibly concurrent, execution.

It should be apparent by now that TGRSs are a powerful *generalised* computational model able to accommodate the needs of a number of languages, often with divergent operational semantics, such as lazy functional languages [24,28], 'eager' concurrent logic languages [20,33], or combinations of them [21]. Furthermore, recent studies have shown that TGRSs are able to act as a means for implementing languages based on computational models such as Linear Logic [5], π-calculus [3,16] and OOP [34]. In addition, the implementation of TGRSs themselves on a variety of (dataflow and graph rewriting) machines such as Alice [12], Flagship [27,43], and GRIP [35] has been extensively studied. Fig. 2 is a modification of the previous one for Dactl. For more information on TGRSs the reader is advised to consult [36,39] whereas for Dactl appropriate references are [17–19].

In the following section we discuss those features of Dactl which we feel have played an important role in the success of the language to act as an interface between (concurrent) declarative languages and parallel machine architectures. Bearing in mind the wide applicability of the language we feel that these features are general enough and so fundamental that all similar intermediate compiler target

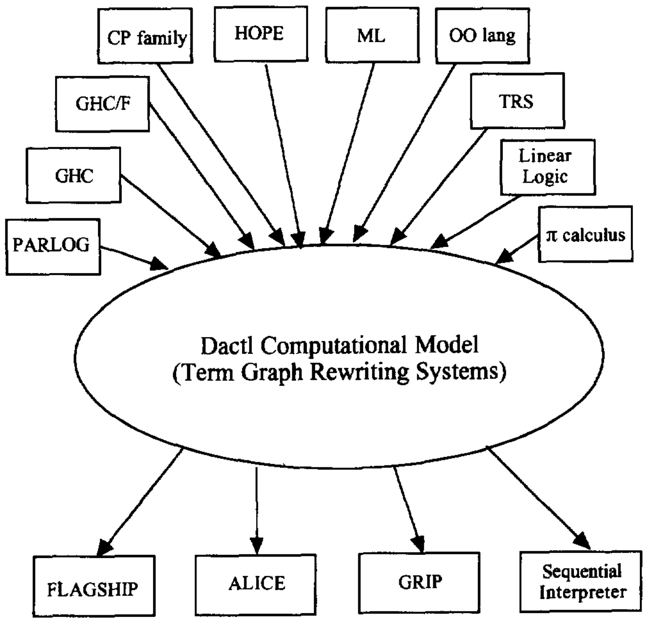


Fig. 2. Dactl as UNCOL.

languages should possess them. We should probably stress the point here that the assessments made in this paper are based on a personal perspective and may not necessarily be shared by the designers of the language or anyone else that participated in the Flagship project.

3. Essential features of a CTL for parallel machines

3.1. Language embedding

The notion of language embedding [38] is essential in understanding the importance of the points raised in this section. In comparing two languages L1 and L2 we say that L1 is more expressive or more general or stronger than L2 if: (i) L1 supports certain programming techniques in a 'better' way than L2 does and (ii) L2 can be 'naturally' embedded in L1.

Since all languages are trivially Turing equivalent we should be aware of the fact that notions such as 'better' or 'naturally' are essentially ad hoc. However, regarding the second point, we can say that a language L2 can be naturally embedded into another language L1 if the main features of L2 can be supported directly by L1 and there is no need to 'program them around'. In other words, an implementation of L2 in L1 should be able to *absorb* L2's main features rather than *reify* them. A typical example is pattern matching; consider the following pieces of code written in some functional language and its translation to Dactl:

F – lang : `p(H : T, g(X)) → q(H, T, X).`

Dactl : `P[Cons[h t] G[x]] ⇒ *Q[h t x];`

We note that in the equivalent Dactl rule the pattern matching is completely absorbed by Dactl's computational model. However, the same cannot also be said about the following case where the initial rule is a Prolog-type clause:

Prolog : `p([H|T]) : – q(H, T).`

Dactl :
```
P[Cons[h t]] ⇒ *Q[h t];
P[v : Var] ⇒ *Q[h t],
v : = *Cons[h : Var t : Var];
```

Since Dactl supports only pattern matching (one-way unification) and not full (two-way) unification, there is a need to work around the case where P is called with an uninstantiated argument (a variable).

So, what are precisely the essential features that a compiler target language should possess such that the mapping of some user-level language onto the CTL is natural and most of the language's functionality is absorbed by that of the CTL (rather than being reified)? Our experience from implementing a number of languages (logic, functional and object-oriented) in Dactl indicates that a CTL should: (i) have operational semantics flexible enough to accommodate in a genuine (as opposed to simulated by 'programming around') way the, often divergent, needs of various programming models (e.g. eager vs. lazy evaluation); (ii) be fine grain enough so that the CTL code produced when some language is mapped onto it can be 'fine tuned' to be rendered more efficient; (iii) be flexible as to what constitutes a 'variable' since the concept of a variable differs significantly from one language to another; (iv) support some degree of atomicity so that one can reason about the CTL's behaviour at run-time by mapping concurrent activities to serialisable actions, but not to an extent that implementing the CTL would demand unacceptably high locking overhead.

In the rest of this section we show that Dactl supports these features, we discuss cases where such support could possibly improve and we illustrate how an implementation of some language onto Dactl can benefit from exploiting the availability of these features to derive more efficient Dactl code.

3.2. Flexible operational semantics

The operational semantics of Dactl which we described in the previous section are fine grain and rather universal. Thus, they allow the direct modelling of more concrete operational semantics as we find them in user-level languages. The following definition in Dactl of an append function illustrates the above points:

```
Append[Nil y] ⇒ *y|
Append[Cons[h t] y] ⇒ #Cons[h *Append[t y]];
```

Note that 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. The above code could be generated if the original program was written in, say, a functional language with strict operational semantics. Nevertheless, the second rule can also be written instead as follows:

```
Append[Cons[h t] y] ⇒ *Cons[h *Append[t y]];
```

This rule specifies an eager evaluation strategy where the partial result of the reduction of Append is made available to its caller while the recursive call is executed in parallel. Furthermore, it is also possible to generate the following encoding:

```
Append[Cons[h t] y] ⇒ *Cons[h Append[t y]];
```

This rule corresponds to a lazy version; the recursive Append will remain dormant until the original caller activates it again. This code could be generated if the original program was written in a functional language with lazy operational semantics.

To help the reader appreciate the significance of this feature, we recall that in eager languages lazy evaluation can only be simulated. In concurrent logic languages [38] this is achieved by means of reversing the producer–consumer relationship; the consumer now produces initially a list of variables (the number of variables signifying the amount of work the producer should produce) and the variables are then bounded by the producer. If the consumer needs further data from the producer, it produces a new list of uninstantiated variables which is then sent to the producer, and so on. This technique, however, is both cumbersome to program (such programs are difficult to be read and understand) but also inefficient (since both the producer and the consumer are continuously active processes consuming machine resources, while the whole concept of lazy evaluation is for producers to remain dormant until needed). Table 1 illustrates the above points.

We have run two Dactl programs producing in a lazy fashion the first 50 Hamming numbers, the first version with simulated lazy evaluation using the above described techniques and the second version using the true lazy evaluation features as supported by the language. The table presents some important statistics as produced by the Dactl implementation. It is clear that the second version is nearly twice as fast compared to the first version.

3.3. Ability to fine-tune the produced compiler target language code

Translating a program written in any (concurrent logic or functional) language to Dactl requires bridging the gap between the high-level operational semantics of the involved language and the 'medium' level one of Dactl. This necessitates applying a number of suitable transformations (and optimisations) to the original program aiming at producing the most efficient Dactl code. The question that arises here is whether those transformations should be done at the level of the language in question or at the Dactl level. Our experience has shown that provided the intermediate representation does not adhere to a specific operational semantics (as it is indeed the case for Dactl) the transformations should be done at the level of the intermediate representation. A typical example is the case of guarded clauses in concurrent logic programs or similar guarded functions in functional programs:

Table 1

Program version	R[a]	PC	AvP	MxP
Simulated lazy	3471	1098	4.98	30
Truly lazy	1202	736	2.60	9

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

Concurrent logic language :

```
H(...) : − G1(...), ..., Gn(...) | B(...).
```

Functional language :

```
H(...) → B(...) if G1(...) and ... and Gn(...);
```

Bearing in mind that Dactl has no concept of guards, the concurrent logic clause (the same argument applies also to the guarded function) could be translated to an equivalent set of Dactl rewrite rules in either of two ways.

(i) Translate the clause into an equivalent clause in flat form:

```
H(...):- New_G1(...,Status1), New_G2(...,Status2),
control(Status1,Status2,Status), commit(Status,...).

control(true,true,Status):- Status = true.
control(false,_,Status):- Status = false.
control(_,false,Status):- Status = false.

commit(true,...):- B(...).
commit(false,...):- false.
```

The transformation of the above flat version to Dactl is straightforward:

```
H[...] ⇒ *New_G1[... status1:Var], New_G2
[... status2:Var],
*Control[status1 status2 status], *Commit[status ...];

Control[True True status:Var] ⇒ *True, status: =
*True;
Control[p1:(ANY-False) p2:(ANY-False) st] ⇒
#Control[p1 p2 st];
Control[ANY ANY status:Var] ⇒ *True, status: =
*False;

Commit[True ...] ⇒ *B[...]|
Commit[False ...] ⇒ *False|
Commit[status:Var ...] ⇒ #Commit[status ...];
```

The problem with the above piece of code is that it is more complicated than it need be. For instance the `Control` process need not commence execution until both the `New_G1` and `New_G2` processes have terminated execution; the same can be said about the `Commit` process. However, it is not possible to express this functionality at the level of the concurrent logic programming language since function composition, which is effectively what is required, is not truly supported by that formalism.

(ii) Alternatively, the transformations can be done at the Dactl level by examining the structure of the original clause and generating a particular kind of rule set for each different case [33]. For the above clause the generated rule set is as follows:

```
H[...] ⇒ #Control[*G1[...] *G2[...] ...];
Control[True True ...] ⇒ *B[...];
Control[p1:(ANY-False) p2:(ANY-False) ...] ⇒
#Control[p1 p2 ...];
Control[ANY ANY ...] ⇒ *False;
```

Table 2

Program version	R[a]	PC	AvP	MxP
Language level transform	300	79	6.04	36
Dactl level transform	92	37	3.81	11

[a] See Table 1 for abbreviations.

Note the reduced number of processes that are generated compared with the previous version. In more complicated cases the gains in code size and number of process reductions are more significant. A number of other optimisations in pattern matching, etc. can also be performed [33].

Again, the reader can appreciate the significance of this feature by noting how efficient is the mapping of a program finding the *N*th element in a binary tree.

The first version was produced by performing the required transformations at the level of the user's language (in this case a typical concurrent logic language such as Parlog [23] or GHC [44]). The second version was produced by performing the required transformations at the Dactl level. It is obvious that the second version is significantly faster than the first one (Table 2).

3.4. Variable representation

We consider the issue of what exactly constitutes a variable at the level of an intermediate representation as being of paramount importance in the successful design of such a formalism. In TGRSs and, indeed, languages based on them such as Dactl, a 'variable' is simply any node which is *overwritable*, i.e. it can be rewritten with a sub-root redirection. Our experience in dealing with concurrent logic, functional and, more recently, object-oriented and linear logic based languages has shown that the representation of a 'variable' object can range from a simple graph node to a small subgraph rooted at such an overwritable node and comprising some very useful information particular to the semantics and characteristics of the language or formalism in question. For instance, to represent variables in languages like Parlog [23] or Strand [15] a simple overwritable graph node suffices. However, in GHC ([44]) a variable in a guard cannot be instantiated by any unify operations other than the ones invoked within the guard. So in the following clause

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

a unify operation invoked in `q` can instantiate the variable `Y` created in `q`'s environment but not the variable `X` which was imported from `p`'s environment. So every time at run-time a unification is about to be performed, the environment of this operation must be checked against the environment where the variable(s) involved in the unification was (were) created. A number of serious problems exist in implementing GHC's run-time safety test [44] and it was eventually dropped from the definition and implementation of the language.

In such a framework a Dactl variable now is of the `form Var[env]` where `env` is a pointer to the environment where the variable was created in the first place [20]; also, every unify operation itself carries a pointer to the environment in which it was invoked. Thus, when unification is about to be performed a pointer comparison of the two environments is performed:

```
Unify[env v : Var[env] value] ⇒ *True, v : = *value;
                    {perform unification
Unify[env1 v : Var[env2] value] ⇒ Unify[env1 ^v value];
                    {else suspend
```

Here we should explain the fact that when the same node ID appears more than once in the LHS of some Dactl rule, it is considered to denote a test for pointer equality. So in the above rule the instantiation of the variable `Var` will be performed if its environment `env` is the same as the environment where the `Unify` operation is invoked (the first argument of `Unify`). Compatibility of the two environments is modelled simply as a pointer equality between the two `env` nodes, which if they point to the same node causes the selection of the first rule. Note that any parallel machine that supports graph rewriting [12,35,36,39,43] implements pointer equality efficiently since graph sharing is a fundamental concept in this computational model.

In GHC/F [21], an extension of GHC with functional capabilities including handling of infinite data structures and lazy evaluation, the graph apparatus modelling a variable is further extended with the variable's *defining function* as for instance in

```
LHS[...] ⇒ *f:Lazy_Producer[... v:Var[env f] ...],
*Eager_Consumer[... v ...];

Eager_Consumer[... v:Var[env f] ...] ⇒
#Eager_Consumer[... v ...], *f;
```

In the above example an 'eager' consumer predicate is waiting to receive as input argument the value of a variable which must be instantiated by a lazy function. This is a typical problem in any logic–functional language with concurrent capabilities and introduces deadlock which is usually resolved by means of static analysis techniques which try to detect at compile-time the producer of every variable and generate suitable code. In GHC/F the deadlock is resolved more effectively at run-time by simply firing the lazy producer of the variable. This is possible because the variable itself holds a pointer to its defining function thus providing a window connecting the consumer with the producer.

Furthermore, an overwritable node can play the role of a metavariable by being instantiated to a function application rather than a data structure. The following piece of code implements a nondeterministic 'commit' operation using such overwritable nodes.

```
Fire_Commit[...] ⇒ commit:Var,
*Guard1[... commit], *Guard2[... commit];

Guardi[successful_match_etc commit:Var] ⇒
*True, commit: = *Body1[...];
Guardi[unsuccessful_match_etc commit:Var] ⇒
*False;
```

In the above piece of code both guard processes are executed concurrently and one of them nondeterministically will assign the metavariable `commit` to the corresponding body. These sorts of rule systems arise when programs written in some concurrent logic or functional programming language are translated to the more restrictive than Dactl computational model MONSTR [2]. It is for these reasons that overwritable nodes in TGRS based languages are often referred to as *stateholders* [2,4,5].

3.5. Atomicity of rewrites

We recall that all rewritings specified in a Dactl rule are performed atomically, so in the following example

```
Test_and_Set[v1:Var v2:Var] ⇒
*True, v1: = *1, v2: = *2;
```

either both `v1` and `v2` have the pattern `Var` and are instantiated at the same time or either of them has already been instantiated in which case the matching should fail. This is a very powerful concept and it can be used to model atomic unification supported mainly by the Concurrent Prolog family of languages [38]. However, implementing such a scheme is quite expensive and computational models like MONSTR restrict atomicity to the case of only a single overwritable node. Although for languages that endorse the so-called eventual publication of unification such as Strand [15] not even atomicity of a single overwritable redirection is required, to model nondeterminism effectively we need to guarantee the support of atomic updating of such a single overwritable node at the Dactl or MONSTR level; otherwise, there is no guarantee that the rule system of `Fire_Commit` in the previous example will behave as expected.

Furthermore, by supporting some degree of atomicity, it is possible to reason about the correctness of the language for the CTL (in this case Dactl or MONSTR) translation and the program's behaviour at run-time, an ability that we feel is quite important for any CTL to possess.

4. Conclusions and related work

In mapping a variety of computational models and languages to an intermediate compiler target language based on TGRSs for parallel machines we have identified a number of features which we believe every such CTL formalism should possess, namely:

- Flexibility of operational semantics. In particular, the operational semantics should be fine grain, universal and be based on a minimum set of primitive actions. The more concrete operational semantics (lazy, eager, strict, parallel, even sequential) of some high-level language can then be directly supported by the CTL.
- The CTL should have a liberal view of what constitutes a variable so that different ways of accessing such a variable can be implemented effectively, including metaprogramming techniques. A variable apparatus should therefore be viewed more like a control primitive (the 'stateholder' point of view). The implementation and use of these stateholders must be supported efficiently by the underlying machine architecture.
- Atomicity should be supported at least up to the level of updating a single elementary node. A stronger notion of atomicity will be difficult to implement efficiently (requiring extensive locking) and will not be needed for many families of languages, but a weaker one will also not be sufficient to model effectively and simply important control concepts (such as semaphore handling and the stateholder functionality). Banach [2] provides an excellent discussion on this point which led to the design of MONSTR, a subset of Dactl.
- The CTL should be based on some theoretically sound computational model rather than being a possibly useful but ad hoc set of add-on primitives as is sometimes the case for some otherwise highly successful proposals [1,10,14]. One advantage here is that one can formally prove the correctness of the translation scheme adopted from some high-level language to the CTL [24,28].

Also, a related issue is at what level should source-to-source transformations be done in order to assist most effectively in the generation of efficient corresponding intermediate code. Provided the intermediate representation has no concrete operational semantics, the transformations should be done at the level of the intermediate representation rather than at the level of the language that is being mapped.

It is difficult to compare our work with similar research since we are not aware of any other intermediate formalisms sharing the same purposes as Dactl. Probably the closest work we could consider is that by Levy and Shapiro [30,31] on using Flat Concurrent Prolog as an implementation language for GHC and CFL, a new concurrent functional language. It is precisely this type of work that brought our attention to the issues discussed in our paper. In [30], the mapping of GHC to FCP raises some problems: the specific operational semantics of FCP does not allow the flexible manipulation of the produced code; also GHC's 'clean' pattern matching is translated into a set of notorious read-only unifications. Even more important is the fact that what is translated to FCP is only the *safe* subset of GHC for which there is no need for the run-time test. In [31] a new concurrent functional language is designed and implemented on top of FCP. Although the language is non-trivial and useful, its design is based on the features supported by an existing underlying high-level abstract machine (the FCP language): it does not support true lazy evaluation as most state-of-the-art functional languages and, as is also the case for the GHC mapping, it models pattern matching with read-only unifications. Although this particular argument is now only of historical value since the subsequent version, FCP(:), does support one-way unification [38], the rest of the problems discussed in this paragraph have yet to be resolved in an elegant and efficient way. Furthermore, by designing a new language based on the features that some intermediate formalism is offering and mapping the language down to it, one is not able to show that the intermediate formalism is universal in some sense, i.e. that it can effectively model and implement a variety of computational models which where designed independently of it.

Other languages that have played the role of an intermediate CTL are Lean [7], which is very close to the functional subset of Dactl and is mainly targeted towards functional languages, and AKL [25], which has also somewhat flexible operational semantics within the framework of concurrent logic languages and has been used as an intermediate CTL for Prolog [9]. This work raises another interesting point in the issue of using an intermediate formalism, this time from the software engineering point of view: it may be worth using an existing efficient implementation environment of some language as the basis for providing a fast and reasonably efficient implementation for another language, especially if the programming environment of the former language supports development tools, etc. Bueno and Hermenegildo [9], for instance, translate a subset of (sequential) Prolog onto (concurrent) AKL, thus exploiting in an independent-AND fashion the concurrency of the AKL model rather than having to design from scratch a parallel WAM machine.

Another interesting dimension to the issue of using an intermediate formalism is the employment of some computational model having some desirable features to act as a 'glue' for code written in other languages. Foster and Taylor [15], for instance, use the concurrent language Strand to glue together pieces of code implementing intensive computations (such as scientific computing applications) written in conventional languages such as C or Fortran. Thus Strand plays a loose role of an intermediate formalism acting as a synchronisation interface between sequential pieces of code, possibly written in different languages, running and communicating concurrently with each other. Foster et al. [14] and Thornley [41] go even further and design modifications to existing languages which adhere to some general computational formalism. In this latter case, the intermediate formalism is not something concrete but rather an enforced programming methodology.

At a lower level it is worth mentioning the work by Kamperman [26] where the TGRS is not used as an intermediate computational formalism but rather as a means of representing data in a way that can be exchanged between different applications and programming environments.

References

[1] S. Ahuja, N. Carriero, D. Gelernter, Linda and friends, IEEE Comput. 19 (1986) 26–34.
[2] R. Banach, MONSTR: Term graph rewriting for parallel machines, in: M.R. Sleep, M.J. Plasmeijer, M.C.J.D. Eekelen (Eds.), Term Graph Rewriting: Theory and Practice, John Wiley, New York, 1993, pp. 243–252.
[3] R. Banach, J. Balazs, G.A. Papadopoulos, Translating the pi-calculus into MONSTR, J. Universal Comput. Sci. 6 (1995) 335–394.
[4] R. Banach, G.A. Papadopoulos, Parallel term graph rewriting and concurrent logic programs, Proc. Parallel and Distributed Processing '93, Sofia, Bulgaria, May 4–7, 1993, North Holland, pp. 303–322.
[5] R. Banach, G.A. Papadopoulos, Linear behaviour of term graph rewriting programs, Proc. ACM Symp. on Applied Computing '95, Nashville, TN, USA, Feb. 26–28, 1995, ACM Computer Society Press, pp. 157–163.
[6] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, M.R. Sleep, Term graph rewriting, Proc. PARLE '87, Eindhoven, The Netherlands, June 15–19, LNCS 259, Springer, 1987, pp. 141–158.
[7] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, M.R. Sleep, Lean: An intermediate language based on graph rewriting, Parallel Comput. 9 (1989) 163–177.
[8] R. Bisiani, A. Forin, Multilanguage parallel programming of heterogeneous machines, IEEE Trans. Comput. C-37 (1988) 930–944.
[9] F. Bueno, M. Hermenegildo, An automatic translation scheme from Prolog to the Andorra Kernel Language, Proc. FGCS '92, June 1–5, Tokyo, Japan, ICOT Publ., 1992, pp. 759–769.
[10] K.M. Chandy, C. Kesselman, Parallel programming in 2001, IEEE Software (Nov. 1991) 11–20.
[11] F.C. Chow, M. Ganapathi, Intermediate languages in compiler construction: A bibliography, SIGPLAN Notices 18 (1983) 21–23.
[12] J. Darlington, M. Reeve, Alice: A multiprocessor reduction machine for the parallel evaluation of applicative languages, Proc. ACM FPLCA '81, New Hampshire, 1981, pp. 65–75.
[13] ESPRIT II EP 2025, European Declarative Systems, Vols. 1-6, 1989.
[14] I. Foster, R. Olson, S. Tuecke, Productive parallel programming: The PCN approach, Sci. Progr. 1 (1992) 51–66.
[15] I. Foster, S. Taylor, Strand: New Concepts in Parallel Programming, Prentice Hall, New York, 1990.
[16] J.R.W. Glauert, Asynchronous mobile processes and graph rewriting, Proc. PARLE '92, Champs Sur Marne, Paris, June 15–18, LNCS 605, Springer, 1992, pp. 63–78.
[17] J.R.W. Glauert, K. Hammond, J.R. Kennaway, G.A. Papadopoulos, Using Dactl to implement declarative languages, Proc. CONPAR '88, Manchester, UK, Sept. 12–16, Cambridge University Press, 1988, pp. 116–124.
[18] J.R.W. Glauert, J.R. Kennaway, M.R. Sleep, Final specification of Dactl, Internal Report SYS-C88-11, School of Information Systems, University of East Anglia, Norwich, UK, 1988.
[19] J.R.W. Glauert, J.R. Kennaway, M.R. Sleep, Dactl: An experimental graph rewriting language, Proc. Graph Grammars and Their Applications to Computer Science, LNCS 532, Springer, 1990, pp. 378–395.
[20] J.R.W. Glauert, G.A. Papadopoulos, A parallel implementation of GHC, Proc. FGCS '88, Tokyo, Japan, Nov. 28–Dec. 2, 1988, Vol. 3, pp. 1051–1058.
[21] J.R.W. Glauert, G.A. Papadopoulos, Unifying concurrent logic and functional languages in a graph rewriting framework, Proc. 3rd Panhellenic Computer Science Conference, Athens, Greece, May 26–31, 1991, Vol. 1, pp. 59–68.

[22] J.A. Goguen, C. Kirchner, J. Meseguer, Concurrent term rewriting as a model of computation, Proc. Graph Reduction Workshop, Santa Fe, Mexico, USA, Sept. 29–Oct. 1, LNCS 279, Springer, 1986, pp. 53–93.

[23] S. Gregory, Parallel Logic Programming in Parlog: The Language and its Implementation, Addison-Wesley, London, 1987.

[24] K. Hammond, Parallel SML: A functional language and its implementation in Dactl, Ph.D. Thesis, School of Information Systems, University of East Anglia, Norwich, UK, published by Pitman Publishers, London, UK, 1990.

[25] S. Janson, S. Haridi, Programming paradigms of the Andorra Kernel Language, Proc. ISLP '91, San Diego, USA, Oct. 28–Nov. 1, MIT Press, 1991, pp. 167–186.

[26] J.F.Th. Kamperman, GEL, a graph exchange language, Technical Report, CWI, Amsterdam, The Netherlands, 1994.

[27] J.A. Keane, An overview of the Flagship system, J. Funct. Progr. 4 (1994) 19–45.

[28] J.R. Kennaway, Implementing term rewrite languages in Dactl, Theor. Comput. Sci. 72 (1990) 225–250.

[29] P. Kornerup, B.B. Kristensen, A.O.L. Madsen, Interpretation and code generation based on intermediate languages, Software Pract. Exper. 10 (1980) 635–658.

[30] J. Levy, E.Y. Shapiro, Translation of safe GHC and safe concurrent Prolog to FCP, in: E.Y. Shapiro (Ed.), Concurrent Prolog: Collected Papers, Vol. 2, MIT Press, 1987, Chapter 33.

[31] J. Levy, E.Y. Shapiro, CFL: A concurrent functional language embedded in a concurrent logic programming environment, in: E.Y. Shapiro (Ed.), Concurrent Prolog: Collected Papers, Vol. 2, MIT Press, 1987, Chapter 35.

[32] K.J. Ottenstein, Intermediate program representations in compiler construction: A supplemental bibliography, SIGPLAN Notices 19 (1987) 25–27.

[33] G.A. Papadopoulos, A fine grain parallel implementation of Parlog, Proc. TAPSOFT '89, Barcelona, Spain, March 13–17, LNCS 352, Springer, 1989, pp. 313–327.

[34] G.A. Papadopoulos, Object-oriented term graph rewriting, International Journal of Computer Systems Science and Engineering, CRL Publs, 1997 (accepted for publication).

[35] S.L. Peyton Jones, C. Clack, J. Salkild, M. Hardie, GRIP: A high performance architecture for parallel graph reduction, Proc. FPLCA '87, Portland, Oregon, USA, Sept. 14–16, LNCS 274, Springer, 1987, pp. 98–112.

[36] M.J. Plasmeijer, M.C.J.D. Eekelen, Functional Programming and Parallel Graph Rewriting, Addison-Wesley, New York, 1993.

[37] P. Quintas, Software engineering policy and practice: Lessons from the Alvey program, J. Syst. Software 24 (1994) 67–88.

[38] E.Y. Shapiro, The family of concurrent logic programming languages, Comput. Surv. 21 (1989) 412–510.

[39] M.R. Sleep, M.J. Plasmeijer, M.C.J.D. Eekelen (eds.), Term Graph Rewriting: Theory and Practice, John Wiley, New York, 1993.

[40] T.B. Steel, UNCOL: The myth and the fact, Annu. Rev. Automated Progr. 2 (1961) 325–344.

[41] J. Thornley, A collection of declarative Ada example programs, Technical Report CS-TR-93-05, Computer Science Department, California Institute of Technology, CA, USA, 1993.

[42] D.H.D. Warren, An abstract Prolog instruction set, Technical Note 309, SRI International, Menlo Park, CA, USA, Oct. 1983.

[43] I. Watson, V. Woods, P. Watson, R. Banach, M. Greenberg, J. Sargeant, Flagship: A parallel architecture for declarative programming, Proc. 15th Annual ISCA, Hawaii, May 30–June 2, 1988, pp. 124–130.

[44] K. Ueda, Guarded horn clauses, D.Eng. Thesis, University of Tokyo, Japan, 1986.

[redacted]

George A. Papadopoulos holds a B.Sc. in Computer Science and Mathematics (1982) and an M.Sc. in Computer Science with Applications (1983), both from the University of Aston in Birmingham, UK, and a Ph.D. in Computer Science (1989) from the University of East Anglia, Norwich, UK. He has participated in and is still actively involved in a number of national and international research programs (Alvey's Flagship, ESPRIT II's EDS and PCA, MEDCAMPUS, etc.). He is currently an Assistant Professor in the Department of Computer Science at the University of Cyprus in Nicosia, Cyprus. His research interests include parallel programming, concurrent object-oriented programming techniques, design and implementation of declarative (concurrent constraint and functional) programming languages and multimedia modelling and synchronisation. Professor Papadopoulos is a recipient of an ERCIM Fellowship Award for 1994–1995 supported financially by the EU's Human Capital and Mobility programme.