Essential Features of a Compiler Target Language for Parallel Machines

George A. Papadopoulos†

Department of Computer Science
University of Cyprus
75 Kallipoleos Str.
Nicosia, P.O.B. 537, CY-1678
CYPRUS
E-mail: george@turing.cs.ucy.ac.cy

Abstract

Term Graph Rewriting Systems (TGRS) have 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 TGRS 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 an intermediate Compiler Target Language (CTL) based on TGRS for mapping a variety of programming paradigms of the aforementioned types onto it, highlighting in the process some of the issues which we feel any such intermediate representation should address and which form effectively a minimum set of features every CTL should possess.

1. Introduction

Term Graph Rewriting Systems ([3]) 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. TGRS has been shown to be a powerful *generalised* computational model able to accommodate the needs of a variety of language families, often with divergent operational semantics, such as lazy functional languages, "eager" concurrent logic languages or combinations of them and concurrent object-oriented languages ([2]). Furthermore, it has been shown to be capable of expressing the functionality of computational models such as Linear Logic and π-calculus (see bibliography in [2]). In addition, the implementation of TGRS themselves, by means of associated CTLs such as Dactl ([4]) and MONSTR ([1]), on a variety of (data-flow and graph rewriting) machines such as Flagship ([10]) has been extensively studied. For more information on TGRS, Dactl and MONSTR the reader is advised to consult references [1,3,4,9] and [2] in these proceedings.

In this paper we discuss some of the issues which we feel they have played an important role in the success of Dactl as an intermediate formalism bridging the gap between user-level languages and (parallel) machine architectures. We believe that these issues are generic to the idea of using intermediate representations (rather than being peculiar to the TGRS model or the Dactl language) and should therefore be addressed by any other such potential intermediate representation formalism. Due to space limitations we refrain from introducing Dactl here and we ask the interested reader to consult the relevant section in [2].

2. Some Important Features a CTL Should Possess

2.1 Language Embedding

The notion of language embedding ([8]) is essential in understanding the importance of the points raised

---

† Temporary attachment: German National Research Centre for Computer Science (GMD-FIRST), Rudower Chaussee 5, D-1199 Berlin, Germany. E-mail: hcm@prosun.first.gmd.de

0-7803-2018-2/95/$4.00 © 1995 IEEE

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 the language'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:Var t:Var],
                    v:=*Cons[h t];
```

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 unistantiated argument (a variable).

In the rest of this section we identify a number of features which we consider it is essential for any intermediate Compiler Target Language for the aforementioned type of languages and architectures to support with "reasonable" efficiency. Also, we show that Dactl indeed supports these features and it is thus able to embed languages that are mapped onto it rather than reify them by programming around the languages' features with elaborate code.

2.2 Flexible Operational Semantics

The operational semantics of Dactl 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 Append's reduction 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.

What is important to note in all the above codings is that the different operational semantics required in each case are modelled directly by means of using appropriately the available control annotations rather than be "programmed around" them.

2.3 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 and MONSTR, 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 ([7]) a simple overwritable graph node suffices. However in the language GHC,

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 a unification is about to be performed at run-time the environment of this operation must be checked against the environment where the variable(s) involved in the unification was (were) created.

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; also, every unify operation itself carries a pointer to the environment in which it was invoked ([5]). 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; if they point to the same node then that causes the selection of the first rule. Note that any parallel machine that supports graph rewriting ([10]) implements pointer equality efficiently since graph sharing is a fundamental concept in this computational model.

In GHC/F ([6]), our own 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];

Guard1[successful_match_etc commit:Var]
    => *True, commit:=*Body1[...];
Guard1[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 sort 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 ([1,7]). It is for these reasons that overwritable nodes in TGRS based languages are often referred to as *stateholders* ([1,7]).

2.4 Atomicity of Rewrites

One of Dactl's main features is that all rewritings specified in a 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 ([8]). 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 ([8]) 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, say, the rule system of `Fire_Commit` in a previous example will behave as expected.

3. Conclusions

In mapping a variety of computational models and languages to an intermediate Compiler Target Language based on TGRS 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. Then the more concrete operational semantics (lazy, eager, strict, parallel, even sequential) of some high-level language can 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). Reference [1] provides an excellent discussion on this point which lead 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 it is sometimes the case for some, otherwise highly successful, proposals (such as Linda-type models). One advantage here is that one can formally prove the correctness of the translation scheme adopted from some high-level language to the CTL.

References

[1] R. Banach, MONSTR: Term Graph Rewriting for Parallel Machines, in [9], pp. 243-252.

[2] R. Banach and G. A. Papadopoulos, A Highly Parallel Model for Object-Oriented Concurrent Constraint Programming, these proceedings.

[3] H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer and M. R. Sleep, Term Graph Rewriting, *PARLE'87*, Eindhoven, The Netherlands, June 15-19, LNCS 259, Springer Verlag, pp. 141-158.

[4] J. R. W. Glauert, J. R. Kennaway, and M. R. Sleep, Dactl: An Experimental Graph Rewriting Language, *Graph Grammars and Their Applications to Computer Science*, LNCS 532, Springer Verlag, 1990, pp. 378-395.

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

[6] J. R. W. Glauert and G. A. Papadopoulos, Unifying Concurrent Logic and Functional Languages in a Graph Rewriting Framework, *3rd EPY Computer Science Conference*, Athens, Greece, May 26-31, 1991, Vol. 1, pp. 59-68.

[7] G. A. Papadopoulos, A Fine Grain Parallel Implementation of Parlog, *TAPSOFT'89*, Barcelona, Spain, March 13-17, LNCS 352, Springer Verlag, pp. 313-327.

[8] E. Y. Shapiro, The Family of Concurrent Logic Programming Languages, *Computing Surveys 21(3)*, 1989, pp. 412-510.

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

[10] I. Watson, V. Woods, P. Watson, R. Banach, M. Greenberg and J. Sargeant, Flagship: A Parallel Architecture for Declarative Programming, *15th ISCA*, Hawaii, May 30 - June 2, 1988, pp. 124-130.