Object-Oriented Term Graph Rewriting

George A. Papadopoulos

Department of Computer Science

University of Cyprus

75 Kallipoleos Str., P.O. Box 537, CY-1678

Nicosia

CYPRUS

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

Abstract

The relationship between the generalised computational model of Term Graph Rewriting Systems (TGRS) and Object-Oriented Programming (OOP) is explored and exploited by extending the TGRS model with records where access to parameters is done by naming rather than position. Records are then used as the basis for expressing object-oriented techniques such as object encapsulation and (various forms of) inheritance. The effect is that TGRS with records can now be used as an implementation model for a variety of (concurrent) object-oriented (functional, logic or otherwise) languages but also as a common formalism for comparing various related techniques (such as different forms of inheritance or approaches for providing solutions to problems caused by the combination of concurrency and interaction between objects).

Keywords: Object-Oriented Programming; Concurency and Parallelism;
Programming Language Extensions; Rewriting Systems.

1. INTRODUCTION

The generalised computational model of Term Graph Rewriting Systems ([3]) has been used extensively as an implementation vehicle for a number of, often divergent, programming paradigms ranging from the traditional functional programming ones ([14,16]) to the (concurrent) logic programming ones ([11,19]). The model is also capable of supporting imperative programming techniques, such as destructive assignment which is needed frequently for modelling object behaviour (say, the changing of an object's state); in addition, the notion of sharing, a fundamental concept in the graph rewriting world, corresponds directly to the notion of reusability. However, there has not been so far a coherent attempt to use TGRS as an implementation model for Object-Oriented Programming techniques.

In this paper we explore and exploit the relationship between TGRS and OOP by means of extending the TGRS framework with records, where access to parameters is done by naming rather than position. Records are then used as the basis for expressing object-oriented techniques such as object encapsulation and (various forms of) inheritance ([4,27]). Furthemore, if we also enforce a programming methodology where all functions defined and used in a program access their parameters via the use of a single record we end up with a framework where all pattern matching and function application is done based on named rather than positional parameters. This enhanced framework of "TGRS with records" (or "TGRS with names") provides a powerful formalism able to play the role of a generalised implementation model for a variety of (possibly concurrent) OOP languages but act also as a common platform for comparing the various OOP techniques and assessing their usefulness with respect to each other.

In particular, using as a vehicle the intermediate compiler target language Dactl ([8,9]) based on TGRS and playing the role of a high-level machine language in the FLAGSHIP project ([15]), part of the Alvey program ([22]), we first show how records can be implemented in the language without the need to extend the semantics of the associated TGR computational model; then, we use records as the basis for implementing a variety of OOP techniques. The rest of the paper is organised as follows: the next section introduces the TGRS model and the language Dactl and the third one extends Dactl with records and introduces the methodology of using solely records for function definition and application; the fourth one discusses a number of ways to model fundamental OOP techniques and the fifth one describes some practical benefits arising from this work. The paper ends with a discussion of current and future research.

2. TERM GRAPH REWRITING SYSTEMS AND THE ASSOCIATED COMPILER TARGET LANGUAGE DACTL

The TGRS model of computation is based around the notion of manipulating term graphs or simply graphs. In particular, a program is composed of a set of graph rewriting rules `L=>R` which specify the transformations that could be performed on those parts of a graph (redexes) which match some `LHS` of such a rule and can thus evolve to the form specified by the corresponding `RHS`. Usually ([3]), a graph `G` is represented as the tuple $<N_G, root_G, Sym_G, Succ_G>$ where:

- N_G is the set of nodes for `G`
- $root_G$ is a special member of N_G, the root of `G`
- Sym_G is a function from N_G to the set of all function symbols
- $Succ_G$ is a function from N_G to the set of tuples $N_G{}^*$, such that if $Succ(N)=(N_1 \ldots N_k)$ then `k` is the arity of `N` and $N_1 \ldots N_k$ are the arguments of `N`.

Note that the arguments of a graph node are identified by position and in fact we write `Succ(N,i)` to refer to the `i`th argument of `N` using a left-to-right ordering. The context-free grammar for describing a graph could be something like

```
graph  ::=     node | node+graph
node  ::=      A(node,…,node) | identifier | identifier:A(node,…,node)
```

where A ranges over a set of function symbols and an identifier is simply a name for some node.

In the associated compiler target language Dactl, a graph G is represented as the tuple $<N_G, root_G, Sym_G, Succ_G, NMark_G, AMark_G>$ where in addition to those parts of the tuple described above we also have:

- $NMark_G$ which is a function from N_G to the set of node markings $\{\varepsilon, *, \#^n\}$
- $AMark_G$ which is a function from N_G to the set of tuples of arc markings $\{\varepsilon, \hat{}\}^*$

A Dactl rule is of the form

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

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 Pattern is of the form $F[x_1:P_1 \;\ldots\; x_n:P_n]$ where F is a symbol name, x_1 to x_n are node identifiers and P_1 to P_n are patterns. In particular a pattern P_i can be, among others, of the following forms with associated meanings:

ANY	it matches anything
INT,CHAR,STRING	with obvious meanings
READABLE	it matches a symbol name which can only be matched
CREATABLE	it matches a symbol name which can be matched and created
REWRITABLE	it matches a symbol name which can be rewritten with root overwrites

`OVERWRITABLE`	it matches a symbol name which can be overwritten with non-root overwrites
(P_1+P_2)	it matches a symbol name which is *either* P_1 *or* P_2
(P_1&P_2)	it matches a symbol name which is *both* P_1 *and* P_2
(P_1-P_2)	it matches a symbol name which is P_1 *but not* P_2

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 `d1` and `d2` 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 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 `p`'s `#`^{n} marking 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.

In order to further illustrate some of Dactl's features which are important for understanding the rest of the paper we present below the equivalent Dactl program for a non-deterministic merge as it would be written in any state-of-the-art concurrent logic language ([24]).

```
merge([X|XS],YS,ZS] :- ZS=[X|ZS1], merge(XS,YS,ZS1).
merge(XS,[Y|YS],ZS) :- ZS=[Y|ZS1], merge(XS,YS,ZS1).
merge([],YS,ZS) :- ZS=YS.
merge(XS,[],ZS) :- ZS=XS.

MODULE Merge;
IMPORTS Arithmetic; Logic;
```

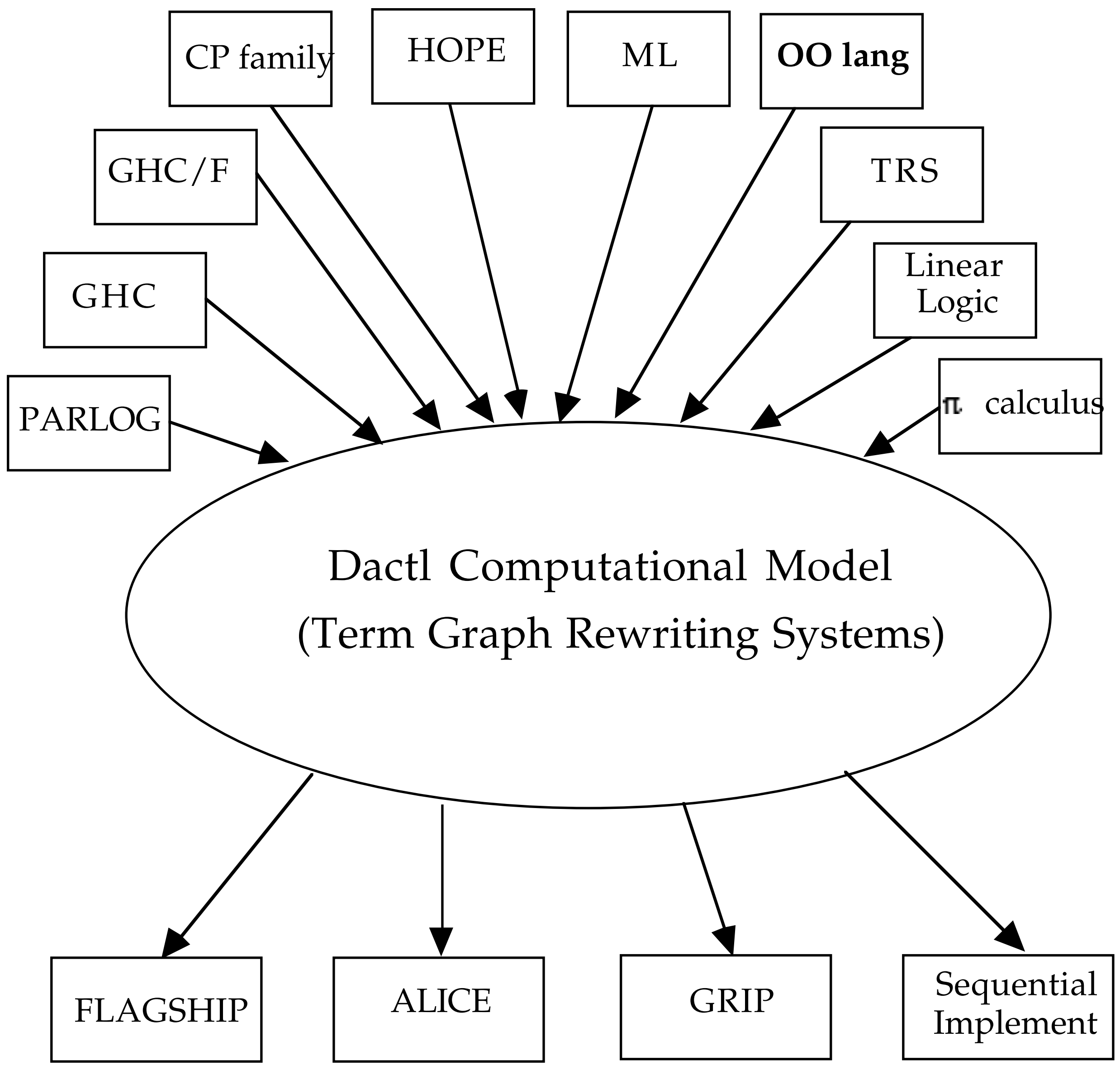
```
SYMBOL REWRITABLE PUBLIC CREATABLE Merge;
SYMBOL OVERWRITABLE PUBLIC OVERWRITABLE Var;
SYMBOL CREATABLE PUBLIC CREATABLE Cons; Nil;
PATTERN PUBLIC PAIR = Cons[head:ANY tail:ANY];
               LIST = (PAIR+Nil);
RULE
     Merge[Cons[x xs] ys zs:Var] => *Merge[xs ys zs1], zs:=*Cons[x zs1:Var]|
     Merge[xs l:PAIR zs:Var] => *Merge[xs l.tail zs1], zs:=*Cons[l.head zs1:Var]|
     Merge[Nil ys zs:Var] => *True, zs:=*ys|
     Merge[xs Nil zs:Var] => *True, zs:=*xs;
     (Merge[p1 p2 p3]&Merge[Var ANY ANY]+Merge[ANY Var ANY]) => #Merge[^p1 ^p2 p3];
     Merge[ANY ANY ANY] => *False;
ENDMODULE Merge;
```

The module starts with a declaration of all the new symbols to be used in the program and the way they are supposed to be used. So, for instance, `Merge` can be rewritten in this module but can only be created in some other module whereas `Var`, playing the role of a "variable", can be overwritten anywhere. Some patterns are also declared: so `PAIR` will match a pattern of the form `Cons[head tail]` where `head` and `tail` can be anything, whereas `LIST` is defined as a pattern which will match either a `PAIR` or the symbol `Nil`.

The first four Dactl rules implement the corresponding ones of the original program. Note again here the use of `=>` instead of `->` for root overwriting and the use of `:=` to model assignment. Note also the selection operator `.` used for illustrative purposes in the second rule where `x.y` is used to refer to a node `y` of some pattern `x`. The fifth rule models the suspension of the process if none of its first two input arguments is instantiated yet; note the use of the pattern operators `&` (conjunction) and `+` (sum). Note here also the use of two notification markings and just one suspension marking. In general, a node of the form $\#P[\hat{}p_1 \ldots \hat{}p_n]$ will be activated in a non-deterministic way when some p_i notifies. In our example this technique models the required non-deterministic merging of the lists. Finally, the last rule reports failure if the input arguments have been instantiated to anything other than a `Cons` or `Nil`. Rules separated by a `|` can be tested in any order whereas those separated by a `;` will be tested sequentially.

We should also stress again the point that the nodes of a graph are labelled with symbols for which an associated access class is specified. In particular, a `REWRITABLE` symbol (such as `Merge`) can be rewritten only by means of ordinary root redirections whereas an `OVERWRITABLE` symbol (such as `Var`) can be rewritten only by means of non-root redirections; also a `CREATABLE` symbol can only be used as the name implies. An overwritable symbol can be "assigned" values by means of non-root overwrites as many times as it is required, and can thus play the role of either a declarative single-assignment variable or the usual imperative one.

It should be apparent by now that TGRS is 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 ([14,16]), "eager" concurrent logic languages ([11,19]) or combinations of them ([12]). Furthermore, recent studies have shown that TGRS are able to act as a means for implementing languages based on computational models such as Linear Logic ([2]) and π-calculus ([7]). In addition, the implementation of TGRS themselves on a variety of (data-flow and graph rewriting) machines such as Alice ([5]), Flagship ([15,30]) and GRIP ([20]) has been extensively studied. Thus, TGRS can be viewed as playing the role of a UNiversal COmputer Language ([29]) for a variety of programming languages and architectures.



For more information on TGRS the reader is advised to consult [21,26] whereas for Dactl appropriate references are ([8,9,10]). The rest of the paper can be viewed as an attempt to add one more box to the top part of the figure shown above, that one of Object Oriented Programming.

3. TERM GRAPH REWRITING WITH RECORDS

In order to be able to model some fundamental OOP techniques we propose the introduction of records where the aim is to support standard record manipulation operations such as record creation, selection and updating of a record's elements, etc. without the need to extend the semantics of the underlying TGR model of computation. In particular, a record is a data structure of the form

```
Record["string_name id"   Name1[value1]   …   Namen[valuen]]
```

where the following symbol declarations have been defined.

```
SYMBOL CREATABLE PUBLIC CREATABLE Record; Unbound;
SYMBOL OVERWRITABLE PUBLIC OVERWRITABLE Name1; … ;Namen;
PATTERN PUBLIC RECORD  =  (Record[STRING]+Record[STRING ANY]+Record[STRING ANY ANY]+
                           Record[STRING ANY ANY ANY]+  …);
                NAME   =  (Name1+ … +Namen);
```

In other words, a record is a data structure comprising a string (its name id), and a sufficient number of overwritable symbols (possibly none if the record denotes a constant value) which represent the fields of the record. In `Name[value]`, `value` can be any value (even another record or a function invocation) including the special symbol `Unbound` with the obvious meaning. The following fundamental operations are allowed on records.

```
CreateRecord[rec_name:STRING Names[name1 … namen] Values[value1 … valuen]]
     => *Record[rec_name Name1[value1]   …   Namen[valuen]];
RecordElement[rec:RECORD name:NAME] => *value;
SetRecordElement[rec:RECORD name:NAME value] => *True, name:=*value;
```

Note that the textual ordering of names and their values in the first rule is immaterial. Note also that in selecting the value of a record element in the second rule, if `Name` is not a field of the record the rule returns `False` as the answer. However, an attempt to update a record field which does not exist (third rule) is considered a null operation; `SetRecordElement` will stil return `True` but of course no field will be updated. This is a typical approach when named parameters or variables are involved in a computation ([4,28]).

All these three operations on records are implemented at a lower level in the language but in a way that adheres to the already established semantics for TGRS. Name comparison, for instance, is done by mapping the respective symbols onto their equivalent string values and checking for string equality. As it has already been mentioned, records in Dactl are updatable objects where the value of a (named) field can be changed multiple times by virtue of the semantics of non-root redirections supported by the underlying TGR model.

The introduction of named parameters in functions by means of using records leads necessarily to more verbose coding. For instance, the following piece of code

```
Sell[rec:RECORD] => #IF[^##AND[^and1 ^and2] new_rec Wrong_Input],
                    and1:#EQ[^*RecordElement[rec Colour] Red],
                    and2:#EQ[^*RecordElement[rec Size] Large],
                    new_rec:CreateRecord["NewRec" names values],
                    names:Names[Colour Size Sold],
                    values:Values[Red Large True];
```

where `IF` and `AND` are defined by the rules

```
IF[True then else] => *then|            AND[True True] => *True;
IF[False then else] => *else;           AND[ANY ANY] => *False;
```

models a function which given a record having named parameters `Colour` and `Size` with values `Red` and `Large` respectively, creates a new record consisting of the fields `Colour` and `Size` having their previous values and a new field `Sold` with value `True`; in any other case the function returns an appropriate error message.

In order to avoid the creation of such elaborate code, we introduce some syntactic sugar. In particular, we define the record field selection operator ◦ whose use is better described by means of the following examples:

```
F[rec:RECORD▫Name:George] => *True, rec▫Age:=*32;
G[rec:RECORD▫Colour:x ans:Unbound] => *True, ans:=*x;
H[rec:RECORD▫Value:x:(ANY-INT)] => *x;
```

The first rule is a candidate for matching if `F`'s argument is a record having a field named `Name` with `George` as value. Upon selection, the rule rewrites to `True` and instantiates the field `Age` of the record to `32`. The second rule is a candidate for matching if `G`'s argument is a record having a field named `Colour`; in this case the rule rewrites to `True` and instantiates `G`'s second argument to `Colour`'s value. Finally, the last rule is a candidate for matching if `H`'s argument is a record having a field named `Value` whose value is anything but an integer; the rule simply rewrites to that value. It is easy to show that all these sugared versions of selecting and updating fields of records require no semantic extensions to the basic TGRS model and can be implemented in terms of the three basic operations on records that were discussed earlier on. Using the ◦ operator then, the `Sell` rewrite rule can be written simpler as follows:

```
r:Sell[rec:RECORD],                   new_rec:<<"NewRec" Colour:col Size:sz Sold:True>>,
rec▫Colour:col:Red,       ->          r:=*new_rec;
rec▫Size:sz:Large
Sell[ANY] => *Wrong_Input;
```

Note the way the LHS of the first rule is formed to cope with a double matching on the record and the use of the new operator <<…>> for providing a shorter way to create a new record. The first rule could also have been written instead as follows:

```
Sell[RECORD["Rec" Colour[col:Red] Size[sz:Large]]]
     => *<<"NewRec" Colour:col Size:sz Sold:True>>;
```

Nevertheless this version destroys the encapsulation offered by named records: the rule must be aware of all the details related to the contents of the record whereas the previous version is valid for *any* record having fields named `Colour` and `Size` with the required values. In the rest of the paper we will be using sugared syntax with the understanding that all programs shown can be rewritten to the core subset of the language without the need to extend semantically the underlying TGRS model.

This point brings us to the second contribution made in this paper (the first being the introduction of records *per se*): if the use of records is coupled with an enforced programming methodology where, with the exception of built-in functions and other similar objects, function parameters are encapsulated within a record and operations such as pattern matching are viewed as field selections, a compact formalism is yielded able to accommodate fully the needs of OOP such as support for software composition. In addition, data abstraction and object encapsulation is respected since selective ("by need") pattern matching does not require knowledge of the complete structure of the object that is being pattern matched. So a user program does not have to rely on knowing information like the arity of a function or the full range of parameters it takes. In addition, reusability comes for free due to the inherent notion of sharing available in a graph rewriting framework.

We show the principles of programming in "TGRS with names" by means of a couple of suitable examples. The first defines the function `Polynomial` with signature `[x:INT,y:INT,a:INT,b:INT,c:INT] -> [res1:INT,res2:INT]` where the parameters `res1` and `res2` will eventually be bound to the results of computing `(a*x+b*y)*(a*x+b*y)` and `(a*x+b*y)*(a*x+b*y+c)` respectively. Note that by returning two results, `Polynomial` belongs more to the realm of (concurrent) logic programming rather than functional programming.

```
SYMBOL REWRITABLE PUBLIC CREATABLE Polynomial;
SYMBOL OVERWRITABLE PUBLIC OVERWRITABLE Var;

r:Polynomial[rec:RECORD],          ->    share:##IAdd[^*IMul[r▫a r▫x] ^*IMul[r▫b r▫y]],
r▫x:INT, r▫y:INT,                        r▫res1:=##IMul[^share ^share],
r▫a:INT, r▫b:INT, r▫c:INT,               r▫res2:=##IMul[^share ^#IAdd[^share r▫c]];
```

```
r∘res1:Var, r∘res2:Var


r:Polynomial[rec:RECORD],                        ->     #r;
r∘a:(Var+INT), r∘b:(Var+INT), r∘c:(Var+INT),
r∘x:(Var+INT), r∘y:(Var+INT),
r∘res1:Var, r∘res2:Var


Polynomial[ANY] => *False;
```

The first rule performs the computation where we note the sharing of computing the common subexpression `(a*x+b*y)` but also the subsequently concurrent evaluation of the two results. The second rule performs the synchronisation needed in case some of the required input parameters have not been instantiated yet. More to the point, what the + patterns in the LHS of the rule say is that if some of the input parameters are `Var` (i.e. still uninstantiated) but the rest are instantiated to a valid value (i.e. an integer) the function suspends. The third rule returns some appropriate value (such as `False`) since when matching reaches this rule it means that some of the record's parameters are instantiated to an invalid value (i.e. some value other than `Var` or integer). Note that the syntactic ordering of referring to some node `r∘x` is immaterial. So the record representing a polynomial may well have more input or ouput parameters and some other function running in parallel with `Polynomial` could be computing concurrently other related results.

The next example encodes some well known list manipulation functions (including a functional variant of the non-deterministic `merge` shown earlier on) using records.

```
SYMBOL OVERWRITABLE PUBLIC OVERWRITABLE Head; Tail;
PATTERN PUBLIC Nil  = Record["Nil"];
               Cons = Record["Cons" Head[ANY] Tail[ANY]];


Length[rec:Nil] => *0|
Length[rec:Cons] => #IAdd[1 ^*Length[rec∘Tail]];
```

```
Length[unb:Var] => #Length[^unb];
Length[f:REWRITABLE] => #Length[^*f];

Append[rec1:Nil rec2] => *rec2;
Append[rec1:Cons rec2] => *<<"Cons" Head:rec▫Head Tail:*Append[rec▫Tail rec2]>>;
Synchronisation rules for Append

Merge[rec1:Cons rec2] => *<<"Cons" Head:rec1▫Head Tail:*Merge[rec1▫Tail rec2]>>|
Merge[rec1 rec2:Cons] => *<<"Cons" Head:rec2▫Head Tail:*Merge[rec1 rec2▫Tail]>>|
Merge[Nil rec2:(Cons+Nil)] => *rec2;
Merge[rec1:(Cons+Nil) Nil] => *rec1;
Merge[ANY ANY] => *False;
Synchronisation rules for Merge
```

A call to `Length`, for instance, could take the following form (where `INITIAL` denotes the first rule to be tried in a Dactl program).

```
INITIAL => *Length[list], list:<<"Cons" Head:1 Tail:<<"Cons" Head:2 Tail:<<"Nil">>>>>>
```

We should stress a rather important point at this stage: the techniques we present for record manipulation in TGRS are independent of whether the programming framework (more precisely, the OO language that will be compiled down to Dactl code) is a functional or a (concurrent) logic one (or even an imperative one). Thus, we have been rather liberal and general in providing synchronisation code for the concurrently executing functions of a Dactl program. For instance, the last two rules in the definition of the `Length` function, which model the required synchronisation of that function with others, cover both the cases where the original program is a functional one (the last rule will fire the argument if it is still a function and wait until its evaluation returns a record) or a logic one (the one but last rule will suspend waiting for the variable to be instantiated to some record by some other producer predicate). Wherever we feel that by refraining to include such code we do not compromise our assertions about the validity of our examples, and in order to keep their length down to resonable size, we will refrain from including it from now on. However, it should be assumed implicit in any complete and formal translation from some OO language to its equivalent Dactl code.

Our final example in this section illustrates the use of TGRS with records to act as an implementation model for "calculi with names" such as λN ([4]) and the γ-calculus ([28]), as it was done for other calculi ([2,7]). For lack of space we only highlight briefly the principles of such a mapping, concentrating on λN. The syntax of λN and the associated reduction rules are as follows:

$x_i \in$ Names, $v \in$ Variables, $a,b \in$ Terms

v	::=	x | \v	*variables*
a	::=	$\lambda(x_1,\ldots,x_n)a$	*abstraction*
	|	v	*variable*
	|	a(x->b)	*bind operation*
	|	a!	*close operation*

$(\lambda(x_1,\ldots,x_n)a)(x_i\text{->}b)$ -> $\lambda(x_1,\ldots,x_n)(a[x_i\text{->}b])$

$(\lambda(x_1,\ldots,x_n)a)!$ -> a

Note that `\v` is a protected variable which belongs to the outer abstraction (use of α-conversion is for obvious reasons not possible). All the names introduced in the definition of an abstraction are at the same level. Note the use of two reduction rules where `a(x->b)` binds the name `x` in `a` to `b` and `a!` performs the actual reduction. This allows the binding of a function's arguments to be done in any order, even concurrently, and the actual function reduction to be done when all (input) arguments have been instantiated, thus achieving some of the functionality of a concurrent logic program.

The principles of mapping λN to TGRS with records are shown below.

```
SYMBOL REWRITABLE PUBLIC CREATABLE L;                { the λN abstraction
SYMBOL OVERWRITABLE PUBLIC OVERWRITABLE X1;…;Xn;     { its names
PATTERN PUBLIC L_REC = Record["L" X1[ANY] … Xn[ANY]];
                 NAME = (X1+ … +Xn);
```

```
RULE
      some_lhs          ->      l:<<"L" X1[Unbound] … Xn[Unbound]>>,
                                              { creating a template for the abstraction
                                l∘Xi:=…, l∘Xj:=…;
                                              { binding some of L's named parameters

      r:L[l:L_REC]      ->      l∘Xk:=…,      binding some more parameters
                                *r;           firing the abstraction
```

Note that what happens in general when an expression `a(x->b)(x->c)` is evaluated, depends very much on the semantics of the underlying computational formalism. In λN itself, the first binding is performed and the second is simply ignored. In other models ([28]) the two bindings could be attempted in a non-deterministic way but again the one that did not manage to update `x` will not produce an error. This functionality can be easily captured in TGRS with records by simply examining whether the target `r` in `r∘x` is still an overwritable node and abandoning the operation otherwise. If a model dictates so, a totally imperative approach can also be modelled where both the above bindings are performed either sequentially or in a non-deterministic concurrent way with the last one prevailing.

4. INHERITANCE MECHANISMS IN TGRS WITH RECORDS

4.1 Object Encapsulation and Representation

The extended with records framework provides the required mechanism needed for object encapsulation and representation and acts as the basis for modelling various forms of inheritance. In general, there are a number of ways to achieve encapsulation (and hence inheritance) either in a (concurrent) logic ([6,13,27,28]) or a functional ([4,17,18,31]) framework. Here we concentrate on those techniques that use records as first class objects.

We use as an example the unavoidable 2- and 3-dimensional point with a number of elementary methods associated with it. A first approach is shown below:

```
SYMBOL CREATABLE PUBLIC CREATABLE SetX; SetY; Move; Clear;              { methods
SYMBOL REWRITABLE PUBLIC CREATABLE Point2D; Point3D;                    { object
SYMBOL REWRITABLE Point2D_aux;
SYMBOL OVERWRITABLE PUBLIC OVERWRITABLE XCoord; YCoord; ZCoord; Colour; { parameters
PATTERN PUBLIC POINT2D = Record["Point2D" XCoord[ANY] YCoord[ANY]];     { classes
               POINT3D = Record["Point3D" XCoord[ANY] YCoord[ANY] ZCoord[ANY]];
            POINT2DCol = Record["Point2DCol" XCoord[ANY] YCoord[ANY] Colour[ANY]];
                 POINT = (POINT2D+POINT3D+POINT2DCol);    { more general class patterns
                  NAME = (XCoord+YCoord+ZCoord+Colour);

Point2D[rec:POINT Nil] => *rec|
Point2D[rec:POINT Cons[message rest]] => #Point2D[^*Point2D_aux[rec message] rest];

Point2D_aux[rec name:NAME] => *rec▫name|
Point2D_aux[rec SetX[dx:INT]] => *rec, rec▫XCoord:=*dx|
Point2D_aux[rec SetY[dx:INT]] => *rec, rec▫YCoord:=*dy|
Point2D_aux[rec Move[dx:INT dy:INT]]
     => *rec, rec▫XCoord:=*IAdd[rec▫XCoord dx], rec▫YCoord:=*IAdd[rec▫YCoord dy]|
Point2D_aux[rec Clear] => *rec, rec▫XCoord:=*0, rec▫YCoord:=*0;
```

A number of points should be cleared before we continue the discussion. The class of a 2- or 3-dimensional point is represented by means of an appropriate record. The corresponding object is represented by means of a recursive function comprising two arguments: the record and a stream parameter accepting a list of messages to this object. Each message in the list is handled by an associated auxiliary function which returns the updated record.

Note that the updating of the record is done "in place", effectively by resorting to imperative techniques (destructive assignment) by means of non-root overwrites supported by TGRS. This is a more efficient approach than creating a brand new record with the new values and does not compromise the declarativeness of the framework. Note also the way class patterns are defined, allowing the use of methods such as `Move` for a variety of similar types of points. This is possible, of course, because the use of records renders the update operations polymorphic and independent of, say, the number and names of arguments in other types of points.

A further couple of points should be noted which apply in fact to all the examples in this section: i) the updating operations of different fields in a record are done concurrently, exploiting here the high degree of fine grain parallelism available in the TGR model; ii) messages like `Move` should, strictly speaking, also be records so that the textual position of their components (and possibly other information such as their number) need not be known by the object handling them. We choose the ordinary positional approach however for the sake of keeping the complexity of the presentable code down to a manageable level.

The above approach achieves object encapsulation and reuse of methods by instances of other similar classes. However an even more flexible approach is the following where the record itself is extended with a `Class` parameter denoting the class of the object and allowing the extracting of a method from a class and its direct application to instances of subclasses. We present only that part of the code which is different from the one shown above.

```
SYMBOL REWRITABLE Point2Da;
PATTERN PUBLIC POINT = Record["Point2D" Class[Point2D] XCoord[ANY] YCoord[ANY]];
              METHOD = (GetX[ANY]+GetY[ANY]+SetX[ANY ANY]+SetY[ANY ANY]+
                        Move[ANY ANY ANY]+Clear[ANY]);

Point2D[method:METHOD] => *Point2D_aux[method]|
Point2D[Nil] => *True|
Point2D[Cons[message rest]] => #Point2Da[^*Point2D_aux[message] rest];

Point2Da[ANY Nil] => *True;
Point2Da[ans Nil] => *ans;   { when the last message will cause an answer
Point2Da[ANY Cons[message rest]] => #Point2Da[^*Point2D_aux[message] rest];
```

```
Point2D_aux[GetX[rec:POINT]] => *rec▫XCoord;
Point2D_aux[GetY[rec:POINT]] => *rec▫YCoord;
Point2D_aux[SetX[rec:POINT dx:INT]] => *rec, rec▫XCoord:=*dx|
Point2D_aux[SetY[rec:POINT dy:INT]] => *rec, rec▫YCoord:=*dy|
Point2D_aux[Move[rec:POINT dx:INT dy:INT]]
     => *rec, rec▫XCoord:=*IAdd[rec▫XCoord dx], rec▫YCoord:=*IAdd[rec▫YCoord dy]|
Point2D_aux[Clear[rec:POINT]] => *rec, rec▫XCoord:=*0, rec▫YCoord:=*0;
```

Note that both the method to be invoked and the record whereupon the method will be applied are now part of the messages an object receives. In order to use this approach the appropriate class must be selected first followed by a selection of the required method. Here we make use of the following "metarule" available in Dactl.

Apply_To[f:FUNCTION_NAME $param_1$ … $param_n$] => *F[$param_1$ … $param_n$];

A typical query now takes the following form (where `MovedPoint` is an instance of `Point2D`).

```
MovedPoint[rec:POINT messages] => #Wait_for_ans[rec ^*Apply_To[rec▫Class messages]];

Wait_for_ans[updated_record True] => *updated_record;
Wait_for_ans[updated_record ans] => *ANS[updated_record ans];

INITIAL => *MovedPoint[rec:<<"Point2D" Class:Point2D XCoord:0 YCoord:0>> messages],
           messages:Cons[SetX[rec 2] Cons[SetY[rec 3] Cons[Move[rec 5 6] Nil]]];
```

Note the sharing of the record by all messages sent to `Point2D`.

4.2 Inheritance

The last technique for object encapsulation and representation discussed above can be used as the basis for modelling inheritance where methods defined in a certain class are applied to instances of some other class. In order to illustrate this ability we define a 3-dimensional point which inherits the methods of its 2-dimensional counterpart (some pattern definitions must be redefined and their new values are also shown below).

```
SYMBOL CREATABLE PUBLIC CREATABLE GetZ; SetZ; Move2D; Move3D;
SYMBOL REWRITABLE PUBLIC CREATABLE Point3D;
SYMBOL REWRITABLE Point3D_aux;
SYMBOL OVERWRITABLE PUBLIC OVERWRITABLE ZCoord;
PATTERN PUBLIC POINT = (Record["Point2D" Class[Point2D] XCoord[ANY] YCoord[ANY]]+
                    Record["Point3D" Class[Point3D] XCoord[ANY] YCoord[ANY] ZCoord[ANY]]);
            METHOD2D = (GetX[ANY]+GetY[ANY]+SetX[ANY]+SetY[ANY]);
              METHOD = (METHOD2D+GetZ[ANY]+SetZ[ANY]+
                        Move2D[ANY ANY ANY]+Move3D[ANY ANY ANY ANY]);
```

Point3D is defined in the same way as Point2D

```
Point3D_aux[GetZ[rec:POINT]] => *rec▫ZCoord;
Point3D_aux[SetZ[rec:POINT dz:INT]] => *rec, rec▫ZCoord:=*dz|
Point3D_aux[method:Move2D[rec:POINT dx:INT dy:INT]] => *Point2D[Move[rec dx dy]]|
Point3D_aux[Move3D[rec:POINT dx:INT dy:INT dz:INT]]
     => *Point2D[Move[rec dx dy]], rec▫ZCoord:=*IAdd[rec▫ZCoord dx]|
Point3D_aux[method:METHOD2D] => *Point2D[method];
```

In the example above, a number of OOP techniques are applied simultaneously: i) `Point3D` renames `Move` to `Move2D`, ii) it inherits its code from `Point2D` in order to implement a `Move3D` operation, iii) it defines new methods for accessing and updating the third argument, and iv) it forwards all the other messages to `Point2D`. This is sufficient for modelling inheritance based on static binding. Note that in the last but one rule, the two activities of the invoking of the methods in `Point2D` and the updating of the third argument by `Point3D` itself, are both done concurrently.

However, it is also possible to model more flexible forms of inheritance based on dynamic binding by noting that: i) instead of mentioning explicitly class (and super class) names we can get them from the record information, and ii) methods such as `Move` and `Move3D` can in fact be implemented in terms of more elementary methods such as `GetX`, etc. defined in the same class. This requires simply the extension of some class with an additional field representing its superclass as follows:

```
Record["Point3D" Class[Point3D] Super[Point2D] XCoord[INT] YCoord[INT] ZCoord[INT]]
```

Now the implementation of `Point3D` is as follows.

```
Point3D_aux[GetZ[rec:POINT]] => *rec◦ZCoord;
Point3D_aux[SetZ[rec:POINT dz:INT]] => *rec, rec◦ZCoord:=*dz|
Point3D_aux[method:Move2D[rec:POINT dx:INT dy:INT]]
     => *Apply_TO[rec◦Super Move[rec dx dy]]|
Point3D_aux[Move3D[rec:POINT dx:INT dy:INT dz:INT]]
     => *Apply_TO[rec◦Super Move[rec dx dy]],
        rec◦ZCoord:=#IAdd[dz ^new_z], new_z:*Apply_TO[rec◦Class GetZ]|
Point3D_aux[method:METHOD2D] => *Apply_TO[method◦Super method];
```

where we rely again on the functionality of the `Apply_To` metarule to create dynamically function applications. Note that the classes involved (`Point2D` and `Point3D`) are not mentioned explicitly and are derived by means of retrieving the appropriate values from the parameters `Class` and `Super`. These classes can, of course, be changed dynamically as in the following example

```
Point[rec:POINT] => *Point3D[rec], rec◦Super:=*Point3DColour;
```

where all references to super will now use the methods in the class `Point3DColour`.

Finally, note that the operational semantics of the inheritance mechanism discussed in this section is not based on code copying as it is done in other models ([13]) and no code duplication is required.

4.3 The Inheritance Anomaly

An important point to note is the fact that by extending the definition of the record with the `Class` and `Super` fields we have actually turned the classes identifiers into first class values; a class identifier can now be matched in the LHS of some rule and its value can even be changed. It has recently been shown ([18]) that this approach can lead to an elegant solution to the so called inheritance anomaly caused by the conflicting nature of concurrency and inheritance ([17,18,31]).

In particular, the problem arises because of the need in a concurrent OOP environment to include in the code for the methods of an object some synchronisation constraints on the acceptance of a message by an object. A typical example is that of a bounded buffer where a `put` message should be accepted only if the buffer is not full and a `get` message should be accepted only if the buffer is not empty. The problem that arises then is that the code which is responsible for performing the checking on whether the current state of a bounded buffer object allows it to accept a `put` or a `get` message, the so called synchronisation code in general, is difficult to inherit without extensive, sometimes, redefinitions. Using only the most important parts of the produced code the `put` and `get` methods of such a bounded buffer object could be defined as follows:

```
PUBLIC PATTERN BUFFER = Record[“Buffer” Class[Buffer] LIMIT[INT] CONTENTS[LIST]
                                                             IN[INT] OUT[INT]];

Buffer[put:Put[ANY buffer:BUFFER]]          { if buffer is full ignore the message
     => #IF[^#ILESS[^*ISUB[buffer▫IN buffer▫OUT] buffer▫LIMIT] Buffer1[put] True];

Buffer[get:GET[ANY buffer:BUFFER]]          { if buffer is empty ignore the message
     => #IF[^*ILESS[buffer▫OUT buffer▫IN] Buffer1[get] True];

Buffer1[Put[item buffer:BUFFER]] => *buffer, buffer▫IN:=*IAdd[buffer▫IN 1],
                                  buffer▫CONTENTS:=*Cons[buffer▫CONTENTS item];

r:Buffer1[Get[toObject buffer:BUFFER]], -> buffer▫OUT:=*IAdd[buffer▫OUT 1],
buffer▫CONTENTS:Cons[item rest]            toObject:=*item, r:=*buffer;
```

where `Put[item]` adds item to the buffer and `Get[toObject]` sends the last item added to the buffer to the object `toObject`. Now consider having a class `Lock` of lockable objects in general; a lockable bounded buffer object then, which ignores `get` or `put` messages when it is locked, could be defined by multiple inheritance from bounded buffers and lockable objects where the standard approach is to add a boolean value attribute to ascertain whether the state of the object is locked or unlocked. It becomes immediately apparent however that although the issue of a buffer being locked is orthogonal to that of receiving `get` or `put` messages, the methods for these two messages must be redefined in order to test the value of the boolean attribute before accepting any of those messages.

The solution proposed ([18]), which our framework is able to implement because of the way classes and superclasses are represented, is to effectively change the class of some object when it becomes locked or unlocked. The way a lockable bounded buffer object then is defined in our model is shown below.

```
Buffer is defined as before

r:Buffer1[Put[item buffer:BUFFER]],      -> buffer▫IN:=*IAdd[buffer▫IN 1],
buffer▫Class:Buffer                         buffer▫CONTENTS:=*Cons[buffer▫CONTENTS item],
                                            r:=*buffer;
Buffer1[Put[ANY buffer]]                 => *buffer;

r:Buffer1[Get[toObject buffer:BUFFER]], -> buffer▫OUT:=*IAdd[buffer▫OUT 1],
buffer▫CONTENTS:Cons[item rest],            toObject:=*item,
buffer▫Class:Buffer                         r:=*buffer;
Buffer1[Get[toObject buffer]]            => *buffer, toObject:=*Error;

r:Lockable[Lock[rec:RECORD]],            -> rec▫Class:=*Locked[class],
rec▫Class:class:(ANY-Locked[ANY])           r:=*rec;

r:Lockable[UnLock[rec:RECORD]],          -> rec▫Class:=*class,
rec▫Class:Locked[class])                    r:=*rec;

PATTERN PUBLIC LCKBUFFER = Record["LockableBuffer" Class[LckBuf] Super[(Buffer+Lockable)]
                                 LIMIT[INT] CONTENTS[LIST] IN[INT] OUT[INT]];

LckBuf[mess:(Get[ANY ANY]+Put[ANY ANY])] => *Buffer[mess]|
LckBuf[mess:(Lock[ANY]+Unlock[ANY])]      => *Lockable[mess];
```

What we have done simply is to test in `Buffer1` whether the `Class` attribute of the record has the expected value. If it does not then the message received (`Put` or `Get`) is ignored and the buffer object is not modified. Similarly, upon receiving a `Lock` or `UnLock` message, the `Lockable` object changes the `Class` field of the object it receives to `Locked[class]` and back to `class` respectively. Note that the `Put` and `Get` method definitions require no modifications whatsoever. A lockable bounded buffer object then simply invokes the appropriate method depending on the messages it received. What has happened effectively is that the synchronisation code has disappeared and its functionality has been absorbed by the pattern matching performed in the rewrite rules.

5. IMPLICATIONS

5.1 Object Representation, Evolution and Communication

In this section we have used records for expressing a variety of different ways to model object encapsulation and representation. Some of them are more declarative than others (i.e. not relying on destructive operations). In addition, they differ in the way they support inheritance mechanisms, i.e. not supporting it at all (first approach), supporting it statically (second approach) or supporting a fully-fledged dynamic functionality (third approach). It is also possible to define other similar (but not identical in functionality) mechanisms. We recall that the framework in which we have introduced record based object-oriented functionality is an intermediate representation and an associated compiler target language into which other programming (at the user level) languages compile. These languages (among the many existing we mention some well known typical representatives such as [6,13,17,18,23,25,27,28]) advocate different approaches in dealing with object encapsulation, inheritance, etc. which vary with respect to issues such as degree of "declarativeness", support for concurrent method invocation, the exact functionality of the inheritance or delegation (see below) mechanism employed, etc. The three different ways we described in the previous section have shown that our generic framework can model these different approaches and the Dactl with records implementation can employ whatever such mechanism the user-level object-oriented language insists on using. For instance, purely declarative languages like [6,23] may want the Dactl implementation to employ a mechanism like the first one whereby the pure declarative approach (completely replacing an updated record with a copy of the new version) may increase the degree of concurrent method invocation and reduce the amount of locking (the object's state, etc.) at the expense of some copying overhead. On the other hand, in languages with coarser grain object-oriented functionality such as [27], where destructive updating of records and dynamic inheritance are supported in a well encapsulated way, an approach similar to the third one may be more appropriate. Furthermore (and bearing in mind the points raised in section 2), by employing all these different mechanisms it is also possible to use our intermediate formalism as a comparison framework able to systematically organise these different approaches, compare them, examine ways of extending them and study the interaction of the extended frameworks with existing constructs and mechanisms.

Another point worth clearing has to do with object representation and evolution. We have deliberately not developed a specific way that defines in some concrete manner how objects are represented, evolve or communicate with each other. The reason is that these rather more specific aspects that completely define some particular object-oriented framework are language specific. We recall from the discussion so far, that our aim is to define a framework able to accommodate the needs of various classes of object-oriented languages while they are being mapped onto an intermediate formalism like Dactl. Our approach is that all these different aspects have a common denominator: a perpetual entity (object, process or whatever) that recurses and in the process it accepts messages, invokes its own methods or delegates requests to other objects, changes its state accordingly and returns its updated version. All these activities can be done by means of recursive rewrite rules handling records like the ones described in this section without any need (or wish) to refine further the framework. In order to explain further the benefits of our approach we focus our attention to the issue of object communication and method invocation. In object-oriented languages based on the functional programming approach (such as [23]), these mechanisms would be employed by means of function applications. It would be possible to enhance our generic framework with techniques developed for implementing traditional (non object-oriented) functional languages in Dactl such as [8,12,14,16]. On the other hand, in object-oriented languages based on the concurrent logic programming approach (such as [6,13,25,27]), these mechanisms would be employed by means of logic variables modelling channels through which messages representing method calls flow. This time it would be possible to enhance our generic framework with techniques developed for implementing traditional (non object-oriented) concurrent logic languages in Dactl such as [2,8,11,12,19]. We are given the opportunity to address further this point promptly.

5.2 Implementation and Performance Aspects

A prototype implementation of the framework developed in the previous section has been implemented on top of the Dactl system ([9,10]). The only aggregate data structure that Dactl supports is arrays and we have used it to represent a record structure. Names are mapped to unique integer values and operations on records are presented to the underlying system as operations on array elements. Thus, our sequential prototype implementation is not terribly efficient but most of the inefficiency stems from this simulation of records-as-arrays. If records are implemented as first class citizens in C, as it is the case for the rest of the Dactl system, then most of the extra overhead of record handling will be removed. However, considering a distributed or parallel implementation of the Dactl system ([2,15,30]) our framework, being enhanced with records and adhering to an object-oriented programming methodology as described in the previous sections, can derive more efficient code regarding load distribution, communication, etc. More to the point, objects are natural distribution units since they represent autonomous entities which communicate by means of message passing; this can be taken into consideration by a Dactl compiler to organise and distribute the work to be performed accordingly. We illustrate the above points by means of the `Point2D` and `Point3D` classes as they would be modelled by a typical concurrent object-oriented logic programming language ([6,13]) using the standard techniques developed by Shapiro and Takeuchi ([25]). We stress the point here that languages like [6,13] simply provide a sugared object-oriented front-end to the underlying concurrent logic framework. The techniques employed by all these languages are simple variants of the basic model described in [25]. According to this model, a class template is reresented by means of a recursive predicate with sufficient numbers of arguments (an object's attributes, internal values, etc.) and an extra variable playing the role of a communication channel between the object and the rest of the world. For every method that the object must handle there are one or more clauses. This model does not normally support true inheritance but only delegation. In such a framework, the `Point2D` and `Point3D` classes could be modelled as follows (only parts of the code are shown for brevity).

```
poin2D(Point,State) :- point2D(Point,0,0,State).

point2D([setX(V)|Rest],X,Y,State) :- point2D(Rest,NX,Y,NState),
                                     NX=X+V, State=[coord(NX,Y)|NState].
point2D([move(V1,V2)|Rest],X,Y,State) :- point2D(Rest,NX,NY,NState),
                                         NX=X+V1, NY=Y+V2, State=[coord(NX,NY)|NState].
…
point2D([],_,_,State) :- State=[].

poin3D(Point,State) :- point3D(Point,0,0,0,State).
```

```
point3D([setZ(V)|Rest],X,Y,Z,State) :- point3D(Rest,X,Y,NZ,NState),
                                       NZ=Z+V, State=[coord(X,Y,NZ)|NState].
point3D([move(V1,V2,V3)|Rest],X,Y,State) :- point3D(Rest,NX,NY,NZ,NState),
                                            point2D([move(v1,v2),X,Y,IState),
                                            update(IState,X,Y,NX,NY,OState),
                                            NZ=Z+V, State=[OState|NState].
…
point3D([],_,_,_,State) :- State=[].
```

Using the non object-oriented Dactl framework, and in particular techniques like the ones described in detail in [11,19], the above fragment of code would generate a Dactl program like the following (only the essential parts of the code are shown below — the detailed compilation techniques are described in [11,19]).

```
Point2D[point state] => *Point2D[point 0 0 state];

Point2D[Cons[SetX[v] rest] x y state:Var] => *Point2D[rest nx y nstate:Var],
                                             nx:*IAdd[x v],
                                             state:=*Cons[Coord[nx y] nstate];
Point2D[Cons[Move[v1 v2] rest] x y state:Var] => *Point2D[rest nx ny nstate:Var],
                                                 nx:*IAdd[x v1], ny:*IAdd[y v2],
                                                 state:=*Cons[Coord[nx ny] nstate];
Point2D[Nil ANY ANY state:Var] => *True, state:=*Nil;

Point3D[point state] => *Point3D[point 0 0 0 state];

Point3D[Cons[SetZ[v] rest] x y z state:Var] => *Point3D[rest x y nz nstate:Var],
                                               nz:*IAdd[z v],
                                               state:=*Cons[Coord[x y nz] nstate];
Point3D[Cons[Move[v1 v2 v3] rest] x y z state:Var]
     => *Point3D[rest nx:Var ny:Var nz nstate:Var],           (***)
        *Point2D[Cons[Move[v1 v2] Nil] x y istate:Var],
        *Update[istate x y nx ny ostate],
        *nz:*IAdd[z v3], state:=*Cons[Coord[nx ny nz] nstate];
Point3D[Nil ANY ANY ANY state:Var] => *True, state:=*Nil;
```

The precise compilation route of programs such as the one shown above is as follows: the original object-oriented program, written by means of techniques such as [6,13], is first translated to an intermediate and pure concurrent logic notation (essentially the framework [25]) and the resulting code is finally translated to Dactl using the techniques [11,19]. By the time the program is represented by means of Dactl rewrite rules, all the knowledge regarding objects, encapsulated states, etc. have been compiled away without, however, the implementation having taken advantage of it. For instance in the rule **(***)** above, it would be beneficial for the compiler to know that new entities introduced in the rhs of the rule like the graph nodes `nx`, `ny` and `nstate` or the process `Update` actually are used locally by the object and, thus, in a parallel realisation of the language (like [2]) they should normally reside in the local memory of the same processing element as the one where the rest of the entities involved in this rule reside. This information, however, can be retained in our enhanced with records framework. The produced code would be similar to the one described in section 4 where, now, we show (by means of only a handful of rules for reasons of brevity) how its run-time behaviour is optimised by means of introducing compiler directives (shown in bold for the sake of clarity) that take into consideration the original object-oriented structure and intended behaviour.

```
Point2D_aux[rec Move[dx:INT dy:INT]]
      => *rec, rec▫XCoord:=*IAdd[rec▫XCoord dx] <PLACENEAR[rec]>,
          rec▫YCoord:=*IAdd[rec▫YCoord dy] <PLACENEAR[rec]>|
…
Point3D_aux[Move3D[rec:POINT dx:INT dy:INT dz:INT]]
      => *Apply_TO[rec▫Super Move[rec dx dy]] <PLACENEAR[rec▫Super]>,
         rec▫ZCoord:=#IAdd[dz ^new_z] <PLACENEAR[rec]>,
          new_z:*Apply_TO[rec▫Class GetZ] <PLACENEAR[rec]>|
```

In code fragments like `m:*Process[…] <PLACENEAR[n]>` the compiler directive `PLACENEAR` is used to show that the graph node `m` representing a piece of computation `Process[…]` (possibly introduced in the rhs of some rule) should be placed in the same processing element as the one that handles the process `n`. It is possible to use other types of compiler directives ([9]) that add extra functionality depending on the structure of the original object-oriented code and the intended behaviour of the objects involved in some computation.

6. CONCLUSIONS. CURRENT AND FUTURE WORK

We recapitulate on the main contributions of the paper. We have extended the framework of TGRS to support record manipulation and we have shown how this extended framework can be used as the basis for modelling typical OOP techniques in a TGR framework. We have also shown that our framework can support concurrency, reuse (sharing) of computation as well as handle various problems associated with the combination of non-determinism and OOP techniques (such as inheritance). We have built a prototype implementation of records in Dactl and we have used it to test the examples presented in the paper.

Such an extended model of TGRS with records can now play the same role that the ordinary TGRS model has played for the past decade with respect to (concurrent) logic ([11,18]) and functional ([14,16]) languages, namely to provide an implementation vehicle for a number of existing OOP languages and models ([4,17,18,23,27]). In addition, as it was again the case for other declarative formalisms ([8]), it is possible to use TGRS as a common basis for comparing various OOP languages but perhaps more importantly for studying and comparing different approaches regarding object encapsulation, inheritance, etc. Finally, new (concurrent) OOP based on TGRS semantics can be designed and implemented on top of existing TGRS languages.

We are currently defining a high-level syntax aiming at reducing program verbosity and we study the inclusion of an appropriate type system. Also, we investigate further the interaction of parallelism and non-determinism with the OOP techniques supported by our model ([1]).

REFERENCES

1. Agha G., Wegner P. and Yonezawa A. (eds.), *Research Directions in Object-Based Concurrency*, MIT Press, Cambridge, Massachusetts, 1993.

2. Banach R. and Papadopoulos G. A., Linear Behaviour of Term Graph Rewriting Programs, *Proceedings ACM SAC '95*, Nashville, TN, USA, Feb. 26-28, ACM Computer Society Press, 1995, pp. 157-163.

3. Barendregt H. P, van Eekelen M. C. J. D., Glauert J. R. W., Kennaway J. R., Plasmeijer M. J. and Sleep M. R., Term Graph Rewriting, *Proceedings PARLE'87*, Eindhoven, The Netherlands, June 15-19, LNCS 259, Springer Verlag, 1987, p. 141-158.

4. Dami L., Software Composition: Towards an Integration of Functional and Object-Oriented Approaches, Ph.D. Thesis, Department of Computer Science, University of Geneva, Switzerland, 1994.

5. Darlington J, and Reeve M., Alice - A Multiprocessor Reduction Machine for the Parallel Evaluation of Applicative Languages, *Proceedings ACM FPLCA'81*, New Hampshire, 1981, p. 65-75.

6. Davison A., POLKA: A Parlog Object-Oriented Language, Ph.D. Thesis, Department of Computing, Imperial College, London, UK, 1989.

7. Glauert J. R. W., Asynchronous Mobile Processes and Graph Rewriting, *Proceedings PARLE'92*, Champs Sur Marne, Paris, June 15-18, LNCS 605, Springer Verlag, 1992, p. 63-78.

8. Glauert J. R. W., Hammond K., Kennaway J. R. and Papadopoulos G. A., Using Dactl to Implement Declarative Languages, *Proceedings CONPAR'88*, Manchester, UK, Sept. 12-16, Cambridge University Press, 1988, p. 116-124.

9. Glauert J. R. W., Kennaway J. R., and Sleep M. R., Final Specification of Dactl, Internal Report SYS-C88-11, School of Information Systems, University of East Anglia, Norwich, UK, 1988.

10. Glauert J. R. W., Kennaway J. R., Papadopoulos G. A. and Sleep M. R., Dactl: An Experimental Graph Rewriting Language, *Journal of Programming Languages,* 1997, (to appear).

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

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

13. Goldberg Y., Silverman W. and Shapiro E. Y., Logic Programs with Inheritance, *Proceedings FGCS'92*, Tokyo, Japan, June 1-5, 1992, Vol. 2, p. 951-960.

14. Hammond K., Parallel SML: A Functional Language and its Implementation in Dactl, Ph.D. Thesis, University of East Anglia, Norwich, UK.

15. Keane J. A., An Overview of the Flagship System, *Journal of Functional Programming 4 (1)*, 19-45 (1994).

16. Kennaway J. R., Implementing Term Rewrite Languages in Dactl, *Theoretical Computer Science 72*, 225-250 (1990).

17. Matsuoka S., Taura K. and Yonezawa A., Highly Efficient and Encapsulated Re-use of Synchronization Code in Concurrent Object-Oriented Languages, *Proceedings 8th OOPSLA'93*, Washington D. C., USA, Sept. 26-28, ACM Computer Society Press, 1993, p. 109-126.

18. Meseguer J., Solving the Inheritance Anomaly in Concurrent Object-Oriented Programming, *Proceedings ECOOP'93*, Kaiserslautern, Germany, July 26-30, LNCS 707, Springer Verlag, 1993, p. 220-246.

19. Papadopoulos G. A., A Fine Grain Parallel Implementation of Parlog, *Proceedings TAPSOFT'89*, Barcelona, Spain, March 13-17, LNCS 352, Springer Verlag, 1989, p. 313-327.

20. Peyton Jones S. L., Clack C., Salkild J. and Hardie M., GRIP - A High Performance Architecture for Parallel Graph Reduction, *Proceedings FPLCA'87*, Portland, Oregon, USA, Sept. 14-16, LNCS 274, Springer Verlag, 1987, p. 98-112.

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

22. Quintas P., Software Engineering Policy and Practice: Lessons from the Alvey Program, *Journal of Systems and Software 24:1*, 67-88 (1994).

23. Sargeant J., Uniting Functional and Object-Oriented Programming, *Proceedings 1st JSST*, Kanazawa, Japan, Nov. 4-6, LNCS 742, Springer Verlag, 1993, p. 1-26.

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

25. Shapiro E. Y. and Takeuchi A., Object-Oriented Programming in Concurrent Prolog, *New Generation Computing 1*, 25-48 (1983).

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

27. Smolka G., The Oz Programming Model, *Computer Science Today*, LNCS 1000, Springer Verlag, 1995, pp. 324-343.

28. Smolka G. and Treinen R., Records for Logic Programming, *The Journal of Logic Programming 18 (3)*, 229-258 (1994).

29. Steel T. B., UNCOL: The Myth and the Fact, *Annual Review in Automated Programming 2*, 325-344 (1961).

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

31. Yonezawa A. and Tokoro M. (eds.), *Object-Oriented Concurrent Programming*, MIT Press, Cambridge, Massachusetts, 1987.