Concurrent Object-Oriented Programming Techniques in a Term Graph Rewriting Framework

George A. Papadopoulos

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

Email: george@turing.cs.ucy.ac.cy

Abstract

The relationship between the generalised computational model of Term Graph Rewriting (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 delegation 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).

Keywords: Term Graph Rewriting; Records; Delegation; Inheritance; (Concurrent) Object-Oriented Programming Techniques.

1. Introduction

The generalised computational model of Term Graph Rewriting Systems ([1]) has been used extensively as an implementation vehicle for a number of, often divergent, programming paradigms ranging from the traditional functional programming ones ([6,8]) to the (concurrent) logic programming ones ([5,10]). 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 (OOP) 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 OOP techniques such as delegation and (various forms of) inheritance ([2,11]). Furthermore, 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 also act 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 ([4,3]) based on TGRS and playing the role of a high-level machine language in the FLAGSHIP project ([7]), 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; the fourth section introduces the methodology of using solely records for function definition and application and the fifth one provides a number of techniques for modelling fundamental OOP techniques such as delegation and inheritance. The paper ends with a discussion of current, future and related research.

2. Term Graph Rewriting Systems and the Associated 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 ([1]), a graph G is represented as the tuple $\langle N_G, root_G, Sym_G, Succ_G \rangle$ 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...N_k) then k is the arity of N and N_1...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 ith 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,
           x1:=y1,...,xi:=yi,
           μ1z1,...,μjzj
```

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).

In order to 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 (constraint) logic programming language.

```
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 first four Dactl rules implement the corresponding ones of the original program. Note 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 (in other words, the first two rules illustrate an explicit and an implicit, respectively, way for performing pattern matching).

The fifth rule models the suspension of the process if none of its first two input arguments is instantiated yet; here note the use of the pattern operators '&' (conjunction) and '+' (sum) where A&B must match both A and B and A+B can match either A or B. Therefore, what this rule says is that if both the two input arguments of Merge are still unistantiated (i.e. Var) then the process suspends.

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. So, in describing the previous rule we said "both" although the rule formulation suggested "either of" because the sequential operator used there guaranteed that if either input argument was instantiated then one of the first four rules would have been matched.

Another thing to note is 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.

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"
       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 (`OVERWRITABLE`) 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 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 still 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 ([2,11]).

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.

4. Term Graph Rewriting with Names

If the use of records is coupled with an enforced programming methodology where 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. A user program, for instance, 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 encoding some well known list manipulation functions 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[^*RecordElement[rec Tail]]];

Append[rec1:Nil rec2] => *rec2|
Append[rec1:Cons rec2] =>
    #Cons[^*RecordElement[rec Head]
       #Append[^*RecordElement[rec Tail] rec2]];
```

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[^record_list],
           record_list:#CreateRecord["Cons"
                       Names[Head Tail]
                       ^#Values[1 ^tail1]],
           tail1:#CreateRecord["Cons"
                       Names[Head Tail]
                       ^#Values[2 ^tail2]],
           tail2:*CreateRecord["Nil"
                       Names[] Values[]];
```

Note in the second rule for `Append` the concurrent building of both the head and the tail of the appended list. The use of names introduces extra verbosity in the code which, however, can be taken care of by means of syntactic sugar extensions. We can define, for instance, a field selection operator '°' and rewrite `Append`, say, as follows:

```
Append[rec1:Nil rec2] => *rec2|
Append[rec1:Cons rec2] =>
    *Cons[rec1°Head *Append[rec1°Tail rec2]];
```

In order to reduce verbosity in the rhs of some rule during the creation of a new record, we can define a "record template" and use '°' again to instantiate its named parameters. So assuming the following list template declaration

```
PATTERN PUBLIC LIST = Record["Cons"
                             Head[Unbound]
                             Tail[Unbound]];
```

the rule for `INITIAL` can be written as follows.

```
INITIAL => *Length[list], list:LIST,
           list°Head:=1, list°Tail:=list1:LIST,
           list1°Head:=2, list1°Tail:=Nil;
```

Other similar syntactic extensions are also possible. For instance, we can write

```
F[rec°X:'A' ...] => ...|
F[rec°Y:(Unbound+INT) ...] => ...;
```

where the pattern matching for F will succeed if its first argument is a record having either a field named X with value 'A' or a field named Y whose value is the symbol Unbound or some integer. All these syntactic extensions can be translated to the core subset of the language using the fundamental operations on records and a sufficient number of auxiliary rules. In the rest of the paper, however, we use this core subset only (even at the expense of rendering sometimes the code less readable) to highlight the fact that everything we do requires no semantic extensions to the underlying TGR model.

5. Delegation and Inheritance in TGRS with Names

We now show the principles of expressing delegation and inheritance in our framework. The first subsection describes the basic strategy for achieving object encapsulation and representation which is subsequently used to model these OOP techniques.

5.1 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 delegation and inheritance. There are a number of ways ([2]) to achieve encapsulation (and hence inheritance) and here for lack of space we concentrate our attention on only two of them; as an example we use as object the unavoidable 2-dimensional point with a number of elementary methods associated with it.

```
SYMBOL CREATABLE
PUBLIC CREATABLE
    SetX; SetY; Move; Clear;          { methods

SYMBOL REWRITABLE
PUBLIC CREATABLE Point2D;             { object

SYMBOL REWRITABLE Point2D_aux;
SYMBOL OVERWRITABLE PUBLIC OVERWRITABLE
    XCoord; YCoord;                   { parameters

PATTERN PUBLIC
    POINT = Record["Point2D"
                   XCoord[ANY]
                   YCoord[ANY]];      { class
    NAME = (XCoord+YCoord);

Point2D[rec:POINT Nil] => *rec|
Point2D[rec:POINT Cons[message rest]] =>
    #Point2D[^*Point2D_aux[rec message] rest]|
Point2D[rec:POINT fun:REWRITABLE] =>
    #Point2D[rec ^*fun]|
Point2D[rec:POINT var:OVEWRITABLE] =>
    #Point2D[rec ^var];
Point2D[ANY ANY] => *False;

Point2D_aux[rec name:NAME] =>
    *RecordElement[rec name]|
Point2D_aux[rec SetX[dx:INT]] =>
    *SetRecordElement[rec XCoord dx]|
Point2D_aux[rec SetY[dy:INT]] =>
    *SetRecordElement[rec YCoord dy]|
Point2D_aux[rec Move[dx:INT dy:INT]] =>
    ##Wait[rec ^update1 ^update2],
    update1:#SetRecordElement[rec XCoord ^new_x],
     new_x:#IAdd[^*RecordElement[rec XCoord] dx],
    update2:#SetRecordElement[rec YCoord ^new_y],
     new_y:#IAdd[^*RecordElement[rec YCoord] dy]|
Point2D_aux[rec Clear] =>
    ##Wait[rec ^update1 ^update2],
        update1:*SetRecordElement[rec XCoord 0],
        update2:*SetRecordElement[rec YCoord 0];
```

A number of points should be cleared before we continue the discussion. A 2-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 the third and fourth rules in Point2D where we model synchronisation. Depending on whether the paradigm is functional or other (concurrent logic say), the stream of messages may be a function application needed to be fired (if the paradigm supports lazyness) or a logical variable waiting to be instantiated by some message producer. The last rule reports failure if the object is invoked with the wrong type of arguments.

In the fourth and fifth rules for Point2D_aux note the following points. First, we use imperative techniques to update the record fields rather than a "more proper" declarative approach where a fresh record is generated and returned with updated fields; this is done, of course, for the sake of efficiency by exploiting the power of repeatable non-root overwrites. Second, the updating operations of different fields in the record are done concurrently, exploiting here the high degree of fine grain parallelism available in the TGR model. The auxiliary function Wait is simply waiting for both updating operations to complete before returning the record as its result. Third, messages like Move should, strictly speaking, also be records so that the textual position of their componens (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.

Finally, note that the above implementation of methods for a 2-dimensional point can also be used for any other type of point (3-dimensional, etc.), simply by extending the pattern definition of POINT.

```
PATTERN PUBLIC POINT = (Record["Point2D"
                        XCoord[ANY] YCoord[ANY]]
                       +
                        Record["Point3D" ...]
                       +
                        Record["Point3DColour" ...]
                       + ...);
```

This is possible 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.

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 CREATABLE PUBLIC CREATABLE
    GetX; GetY; SetX; SetY; Move; Clear;

SYMBOL OVERWRITABLE PUBLIC OVERWRITABLE
    Class; XCoord; YCoord;

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


Point2D[method:METHOD] => *Point2D_aux[method] |
Point2D[Nil] => *Nil |
Point2D[Cons[message rest]] =>
    #Point2D[^*Point2D_aux[message] rest] |
Point2D[fun:REWRITABLE] => #Point2D[^*fun] |
Point2D[var:OVEWRITABLE] => #Point2D[^var];
Point2D[ANY] => *False;

Point2D_aux[GetX[rec:POINT]] =>
    *RecordElement[rec XCoord] |
Point2D_aux[GetY[rec:POINT]] =>
    *RecordElement[rec YCoord] |
Point2D_aux[SetX[rec:POINT dx:INT]] =>
    *SetRecordElement[rec XCoord dx] |
Point2D_aux[SetY[rec:POINT dy:INT]] =>
    *SetRecordElement[rec YCoord dy] |
Point2D_aux[Move[rec:POINT dx:INT dy:INT]] =>
    ##Wait[rec ^update1 ^update2],
    update1:#SetRecordElement[rec XCoord ^new_x],
     new_x:#IAdd[^*RecordElement[rec XCoord] dx],
    update2:#SetRecordElement[rec YCoord ^new_y],
     new_y:#IAdd[^*RecordElement[rec YCoord] dy] |
Point2D_aux[Clear[rec:POINT]] =>
    ##Wait[rec ^update1 ^update2],
        update1:*SetRecordElement[rec XCoord 0],
        update2:*SetRecordElement[rec YCoord 0];
```

Note that 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 param1 ... paramn] =>
    *F[param1 ... paramn];
```

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

```
MovedPoint[rec:POINT messages] =>
    #Apply_To[^*RecordElement[rec Class]
             messages];
INITIAL => #MovedPoint[^*CreateRecord["Point2D"
                       Names[Class XCoord YCoord]
                       Values[Point2D 0 0]]
                       messages];
```

where we assume that `messages` denotes a list of messages where their first argument is the record to be updated.

5.2 Delegation

Delegation using records can be expressed by representing the values of some of its parameters as selection functions of fields in other records as in the following example.

```
Delegator[rec:RECORD] =>
    *CreateRecord["Delegator" Names[X Y Z]
                  Values[2 5 RecordElement[rec Z]]];
```

where the value of the `Z` field in `Delegator` is taken to be that of the corresponding field in `rec`. Dynamic delegation involving a variable set of named parameters is also possible although some preprocessing is required. In the following delegation construct

```
Delegator[rec:RECORD] =>
    *CreateRecord["Delegator" Names[X Y Other]
          Values[2 5 RecordElement[rec Other]]];
```

`Other` is a special name which stands for all names involved or a subset of them. There is the need to enhance the functionality of the basic record manipulation operations so that they can substitute `Other` with the right name parameter but since the set of these names is finite this can be done with reasonable programming effort.

5.3 Inheritance

The second technique for object encapsulation and representation discussed in section 5.1 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 the same as Point2D

```
Point3D_aux[GetZ[rec:POINT]] =>
    *RecordElement[rec ZCoord]|
Point3D_aux[SetZ[rec:POINT dz:INT]] =>
    *SetRecordElement[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]]
    => ##Wait[rec ^update12 ^update3],
       update12:*Point2D[Move[rec dx dy]],
       update3:#SetRecordElement[rec ZCoord
                                      ^new_z],
       new_z:#IAdd[^*RecordElement[rec ZCoord]
                                       dz]|
Point3D_aux[method:METHOD2D] => *Point2D[method];
```

In the example above, `Point3D` renames `Move` to `Move2D` and inherits its code from `Point2D` in order to implement a `Move3D` operation, defines new methods for accessing and updating the third argument, and delegates all the other messages to `Point2D`. This is sufficient for modelling inheritance based on static binding. 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. So assuming that `Point3D` is now of the form

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

its implementation can be defined as follows.

```
Point3D_aux[GetZ[rec:POINT]] =>
    *RecordElement[rec ZCoord]|
Point3D_aux[SetZ[rec:POINT dz:INT]] =>
    *SetRecordElement[rec ZCoord dz]|
Point3D_aux[method:Move2D[rec:POINT dx:INT
                                    dy:INT]]
    => #Apply_TO[^*RecordElement[rec Super]
                 Move[rec dx dy]]|
Point3D_aux[Move3D[rec:POINT dx:INT dy:INT
                                    dz:INT]]
    => ##Wait[rec ^update12 ^update3],
       update12:#Apply_TO[^*RecordElement[rec
                                    Super]
                          Move[rec dx dy]],
       update3:#SetRecordElement[rec ZCoord
                                      ^new_z],
       new_z:#IAdd[^#Apply_TO[^*RecordElement[rec
                              Class] GetZ] dz]|
Point3D_aux[method:METHOD2D] =>
    #Apply_TO[^*RecordElement[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],
    *SetRecordElement[rec Super Point3DColour];
```

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

Admittedly, the programs produced are verbose but they can be viewed as being written in the kernel form of the language, the high level one being similar to the syntactic sugaring discussed at the end of section 4.

6. Conclusions; Current, Further and Related Work

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 OOP techniques such as delegation and inheritance in a TGR framework. Thus, such an extended model of TGRS with records (or TGRS with names) can play the same role that the ordinary TGRS model has played for the past decade wrt (concurrent) logic ([5,10]) and functional ([6,8]) languages, namely to provide an implementation vehicle for a number of OOP languages. In addition, as it was again the case for other declarative formalisms ([3]), 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.

The use of named parameters as the basis for providing OOP capabilities has been used in a number of computational models ranging from (constraint or otherwise) logic ([11]) to functional ([2]) paradigms. In fact, the work reported in [2] which introduces λN (λ-calculus with names) was the initial inspiration for conducting this research.

In the more general setting of Rewriting Systems, one must mention the work of Meseguer ([9]) who defines a theory of concurrent objects within the framework of the concurrent functional object-oriented language Maude, an extension of OBJ. The developed framework is very powerful but the underlying execution model is multiset rewriting which requires the use of a special Rewrite Rule Machine using associative memory. Also, Maude seems to allow multi-headed lhs of rules which is a very powerful albeit difficult to implement formalism. Our approach, driven from practical considerations, is more modest, admittedly narrower in scope, but also easier to implement.

We are currently defining a high-level syntax aiming at reducing program verbosity and we study the extension of the current framework with a suitable type system. Also, we

investigate the interaction of parallelism and non-determinism with the OOP extensions.

References

[1] H. P. Barendregt, M. C. J. D. 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.

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

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

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

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

[6] 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, 1990.

[7] J. A. Keane, An Overview of the Flagship System, *Journal of Functional Programming*, Vol. 4 (1), pp. 19-45, January 1994.

[8] J. R. Kennaway, Implementing Term Rewrite Languages in Dactl, *CAAP'88*, Nancy, France, March 21-24, LNCS 299, Springer Verlag, pp. 117-131; extended version in *Theoretical Computer Science*, Vol. 72, 1990, pp. 225-250.

[9] J. Meseguer, A Logical Theory of Concurrent Objects, *OOPSLA/ECOOP'90*, Ottawa, Canada, Oct. 21-25, ACM/SIGPLAN, pp. 101-115.

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

[11] G. Smolka and R. Treinen, Records for Logic Programming, *The Journal of Logic Programming*, Vol. 18 (3), April, 1994, pp. 229-258.