Design and Implementation of an Interpreter Engine for a Visual Logic Programming Language

Omid Banyasad Philip T. Cox

{banyasad,pcox}@cs.dal.ca

Faculty of Computer Science, Dalhousie University

Halifax, Nova Scotia, Canada

Abstract. The design and implementation of an interpreter engine for Lograph, a general purpose visual logic programming language, is discussed. The engine takes full advantage of an efficient implementation of Prolog, while providing for animation of execution. The engine is an interpreter that operates on a Prolog translation of Lograph programs and queries. The translated Lograph programs are probed with instrumentation code at appropriate places so that applications of Lograph rules are reported to the visual interface of Lograph as a side effect of the execution of a program.

1 Introduction

Language for Structured Design (LSD), is a visual language for designing structured objects [4]. The goal of LSD is to provide a basis for Computer-Aided Design (CAD) systems which unify the design and programming activities necessary for creating complex, parametrised objects. This is in contrast to existing CAD systems in which design components are built in a 3D modelling environment, and the programming necessary to create parametrised structures is done in a separate textual programming environment. LSD is based on Lograph, a visual logic programming language [3]. Hence, in order to implement LSD we must first implement Lograph.

Although various visual logic programming languages have been proposed and studied, such as VLP [9], VPP [12], TPM [6], CUBE [10], SPARCL [16] as well as some unnamed systems [11,13,14], Lograph has some properties that make it suitable for logic programming in general and as the basis for a design language in particular. First, the semantics can be realised as graph transformations: second, unification is replaced by two execution rules that reveal the details of unification rather than treat it as one large step [8]. Together, these properties allow an execution to be viewed as a movie depicting the morphing of a query into a result, an important property for a design language, where animating the assembly of objects is a desirable feature.

Although the animation of execution of a logic program is not unique to Lograph, for example Pictorial Janus [7] also provides animated execution of programs, what is appealing about Lograph's execution is that both data and operations on them are treated consistently.

In this paper, we report on a design and implementation of an efficient interpreter engine for Lograph. We will identify problems that arise when using Prolog for interpreting Lograph programs and propose solutions for them. We will discuss how this can be achieved by probing Prolog translations of Lograph programs at appropriate places so that applications of Lograph rules can be reflected in the visual representation of a program. Our interpreter will take advantage of the search and backtracking mechanisms of Prolog while implementing the delete and merge rule exclusively.

2 Lograph Syntax and Semantics

Our presentation in this section is based on [3], to which we refer the reader for a thorough description.

2.1 Lograph Syntax

A Lograph *program* is a collection of literal definitions with no terminals in common. A *literal definition* (or definition for short) is a set of cases. A *case* consists of a name, a head and a body. The *head* of a case is an ordered list of n terminals, where n, called the *arity* of the case, is an integer ≥ 0. The *body* of a case is a set of cells, where a *cell* is either a function cell or a literal cell. If a terminal occurs more than once in a case, either in the head or in any of the cells, the occurrences are connected by lines called *wires*.

A *function cell* consists of a *name*, a *root terminal* and a list of *terminals* of length $n \geq 0$, where n is called the *arity* of the cell. A function cell is represented by an icon bearing the name, having a curved face with the root terminal on it, and a flat face along which the terminals, represented by small circles, are arranged. A function cell can have two orientations as shown in Figure 1 (a). Regardless of the orientation, the terminals of the function cell are ordered from left to right. A function cell with arity 0, also called a *constant*, has the simpler representation <name> or <name>, where <name> is the name of the function.

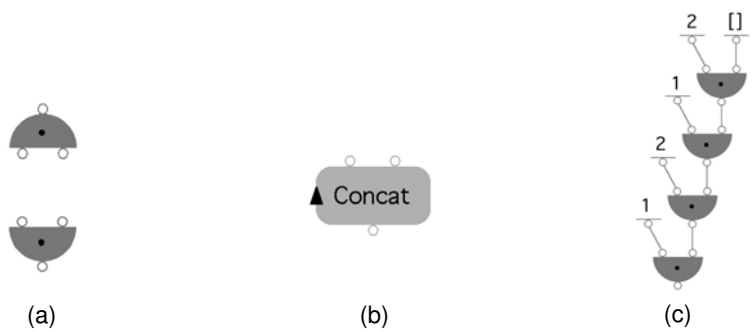

Figure 1 Function cells, a literal cell, and a list in Lograph

In Lograph, as in other logic programming languages, function cells can be used to create data structures. For example, an empty list is represented by a constant, and a non-empty list by a function cell of arity 2 with the first list element attached to the left terminal and the tail list attached to the other terminal. For example, in Figure 1 (c) the constant [] represents the empty list, and function cells named • are used as the list constructors. This graph represents the list[1,2,1,2]. In Lograph, a list can be abbreviated to a constant. For example, the list in Figure 1 (c) can be denoted [1,2,1,2] .

A *literal cell* consists of a name and a list of *terminals* of length $n \geq 0$ called the *arity* of the cell. A literal cell is represented by a rounded rectangle with the name of the literal in the centre. The terminals of a literal cell are arranged along the perimeter starting from the *origin,* a clockwise-pointing arrowhead which may be placed anywhere on the perimeter of the cell. Figure 1 (b) shows a literal cell named Concat with arity 3.

Figure 2 below depicts a definition Concat consisting of two cases.

2.2 Lograph Transformation Rules

Lograph is a visual representation of flat Horn clauses, a specific form of Horn clauses in which equality literals are introduced to remove nested terms. For example, the two cases in Figure 2 represent the two flat Horn clauses:

```
concat(X,Y,Y):-
            X=[].
concat(X,Y,Z):-
            X=dot(H,T),Z=dot(H,Temp),
            concat(T,Y,Temp).
```

We will describe Figure 2 in more details in Section 4.

The semantics of flat Horn clauses are provided by *Surface Deductions* set of three deduction rules discussed in detail in [8]. The three execution rules of Lograph are the pictorial manifestations of the Surface Deduction rules.

Executing a Lograph program involves applying these execution rules to a *query,* which is a network of cells with wires connecting their terminals.

The *Replacement* rule replaces a literal cell in the query with a copy of the body of one of the cases of the definition with same name and arity as the literal cell, if such a definition exists. The corresponding terminals of the head of the case and the terminals of the literal are connected in the process. By *connecting* two terminals, we mean that every occurrence of one of the terminals is replaced by a new occurrence of the other.

The *Merge* rule can be applied to two *compatible* function cells with the same root terminal, where two cells are compatible if they have the same name and arity. First the corresponding terminals of the two function cells are connected, then one of the cells is deleted.

The *Deletion* rule applies to a function cell with a dangling root terminal, simply removing the cell from the query. A terminal is *dangling* if it has no other occurrences.

Since a query provides the starting point for execution of a program, we use the phrases "execution of a program" and "execution of a query" interchangeably.

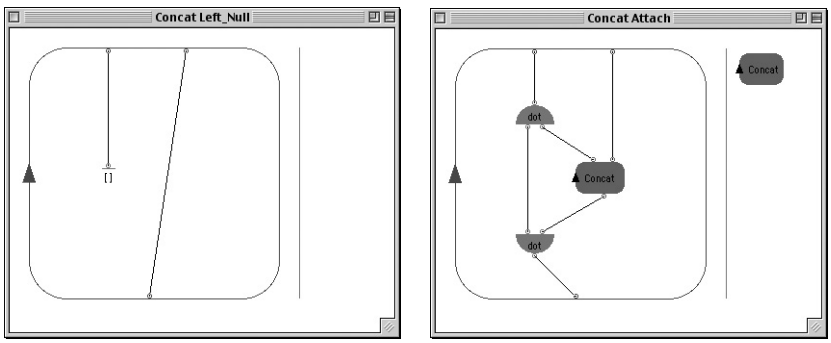

Figure 2 Definition Concat

2.3 Lograph and Prolog

Since we will make frequent comparisons to Prolog, in this section we draw parallels between features of the two languages.

A Lograph definition is analogous to a set of Prolog clauses that define a predicate. A case is analogous to a clause. A literal cell is analogous to a literal in the body of a clause or a query, and a function cell corresponds to a term. A terminal occurring in a literal

cell or in the head of a case corresponds to a variable. A dangling terminal that is not the root of a function cell is analogous to an anonymous variable in Prolog.

Just as the execution of a query in Prolog aims to produce the empty clause, the goal of a Lograph execution is to reduce a query to an empty graph. In Prolog, a failure occurs when two variables cannot be unified because they are bound to terms that begin with different function symbols. The analogy in Lograph occurs when the query is transformed into a graph containing two function cells with a wire connecting their roots, and different names or arities. Such cells cannot be merged, and therefore prevent the query from being transformed into the empty graph. Note that in Lograph, there are other configurations that will have the same effect. The simplest example is a function cell, the root terminal of which also occurs as another terminal of the same cell. This corresponds to a cycle detected by full unification, not usually detected by Prologs.

3 Ordering The Execution Rules

In Lograph there are three sources of non-determinism, the choice of which execution rule to apply, which cell or collection of cells to apply it to, and for the replacement rule, which case of a definition to use. To make Lograph viable as a programming language, we must impose restrictions similar to those imposed on general, first-order, Horn Clause resolution to obtain Prolog.

In Prolog, the clauses that define a predicate are linearly ordered, indicating the order in which they will be applied to a query literal. Similarly, the literals in the body of a program clause are linearly ordered, and once introduced into a query, are executed in that order. The order in which the search space is traversed is therefore well defined, a fact which is exploited by the Prolog programmer.

Clearly, since Lograph is a first-order Horn Clause language like Prolog, we can aim for the same kind of implementation based on depth-first search with backtracking instigated by failure, where failure in Lograph is defined by the presence of undeletable function cells.

As in Prolog, we need to impose two orderings on a Lograph program to obtain a well defined traversal of the search space: specifically, the order in which cases of a definition should be tried in applying the replacement rule, and the order in which the literal cells in a query should be replaced. In addition, we need to decide on the order in which the three execution rules are to be applied. We address the latter issue first.

In [2] we have shown that if a particular ordering of the transformation rules leads to an execution that reduces a query to an empty graph, then any ordering will do the same. Note that these issues do not arise in Prolog since Prolog has only one execution rule.

Just as the Prolog programmer orders the clauses of a predicate definition, the Lograph programmer needs to order the cases of a definition. Since Lograph is a visual language, the sequentiality of text cannot be used as the means to specify this order. As described in Section 4, the Lograph environment provides the programmer with tools for ordering cases.

Since merge and deletion can be applied at any time, the only cells that need to be ordered for execution are literal cells. In Lograph, since the wires are not data flow links, no suitable order can be automatically generated. It is therefore up to the Lograph programmer to specify an order that will do. The solutions we have adopted in the Lograph environment are described in the next section.

4 Programming Environment

Figure 2 illustrates two case definition windows of a literal named Concat. Each case window has a workspace identified by a rounded rectangle to the left and a layer list to the right. The workspace and layer list provide two representations of literal-cell order-

ing, analogous to the two representations of multi-layered images in Photoshop [1]. The layer list is a list of icons similar to the list of layers displayed in Photoshop's "layers" palette. The workspace displays the cells of the case as a series of layers, like the layers in a Photoshop image window which each contain some of the items that make up the whole image. In Lograph, each layer contains one or more literal cells. Cells in the top layer are to be executed first, followed by those in the next layer, and so forth. There is no ordering imposed on literals within a layer, so the programmer can group together literals which could be executed in parallel.

As a clue to the ordering of layers, the literal cells are painted in a range of shades of one colour, the darkest shade applied to the top layer and the lightest to the bottom. The literal and function cells are transparent, giving a sense of depth to the layers. This is similar to the use of transparency to give the illusion of depth in the Macintosh OS X Aqua interface.

Layers are reordered by dragging their iconic representations in the list view. The relative shading of layers is adjusted whenever layers are reordered or a new layer is added. Literals can also be moved from layer to layer. When the number of layers increases, the range of shades is subdivided, resulting in less differentiation between layers. Clearly, as the number of layers grows, the programmer may have to rely more on the layer list for ordering.

Note that during execution, when a literal cell is replaced by a copy of the body of a case, the layers of the case body are placed in front of the existing layers of the query.

Queries are created and executed in a query window, which is similar to a case window, providing both workspace and layer list. A query can be executed in three different modes: **Run**, **Single Step** and **Animated**. In the **Run** mode, the result is computed by the Lograph interpreter and displayed in the query window.

In **Animated** mode, the interpreter displays an animation of each rule application. The deletion of a function cell is animated by fading out the cell and releasing all attached wires, which shrink away from the disappearing function cell towards their other ends. The merge rule is animated by morphing two function cells into one. Animation of replacement is accomplished by expanding the replaced literal to the size of the case that replaces it, then fading in the body of the case together with the necessary connecting wires.

In **Single Step** mode, the interpreter animates each step but stops between steps. In both **Animated** and **Single Step** modes, backtracking is visualized by reversing the animations. A more detailed description of the Lograph programming environment can be found in [2].

5 Interpreter Engine

Our first attempt to implement Lograph was a proof of concept prototype, the core of which was a Java implementation of a standard logic programming interpreter. This had the usual advantages, such as ease of development and debugging, cross-platform executability and so forth. It also had the usual disadvantages, such as slow execution. Once past the proof-of-concept stage, we required a more "industrial strength" implementation, using more appropriate technologies and implementation techniques to obtain a fast and capable interpreter engine. Note that although a fast interpreter is not required when queries are executed in **Single Step** or **Animated** mode, which are mostly for debugging purposes, a fast interpreter for **Run** mode is still desired.

We chose the SICStus implementation of Prolog [15] as the basis for Lograph's interpreter. The release of SICStus Prolog current at the time of writing (version 3.10.0) provides a bidirectional interface to Java, as well as several constraint solvers.

Although flat Horn clause programs can be correctly executed by Prolog, the fine-grained view of unification provided by the surface deduction rules is lost. Consequent-

ly, direct Prolog execution of the flat clauses corresponding to a Lograph program cannot be used for the animated executions described above. This kind of execution is achieved by augmenting the flat clauses with "probes", inserted at appropriate places, so that during the execution of a query, the interpreter can report to the graphical interface, implemented in Java, the actions taken by Prolog that can be interpreted as equivalent to Lograph execution rules. Our interpreter implements the merge and deletion rules explicitly, while search, backtracking, and replacement are provided by Prolog.

5.1 Translating Lograph to Prolog

In the above, we have alluded several times to the correspondence between Lograph and Prolog. In this section we describe how Lograph programs are transformed into Prolog programs. The transformation produces flat Horn clauses which can be directly executed by Prolog.

To illustrate the translation process, we begin with a Lograph query. The textual representation of a query is a flat Horn clause consisting of a set of flat equalities of the form $x = f(\ldots)$ where x is a variable, and a list of flat literals $p_1(\ldots)$,$p_2(\ldots)$,$p_3(\ldots)$, … ,$p_m(\ldots)$, where each … stands for a list of variables. The order of the literals in the clause reflects the order of the corresponding literal cells in the layered structure of the Lograph query. The equalities are not ordered in any particular way since they represent function cells, and can be inserted anywhere in the clause. However, it would be natural to place the equalities at the beginning so merge and deletion can be applied to them before any application of replacement. Although this need not be the case, it will result in a more efficient execution by simplifying the query as soon as possible.

Based on the above, we start with the following Prolog query corresponding to the Lograph query.

```
eqList,p1(…),p2(…),p3(…),…,pm(…).
```

where `eqList` is a list of equalities and `p1(…)` to `pm(…)` correspond to the literal cells in the query.

The body of the clause that represents a Lograph case has structure similar to that of a query. Translating a definition `p` to Prolog, produces a sequence of clauses with the following structure.

```
p(…):-
     eqs1,p11(…),p12(…),…,p1k1(…).
p(…):-
     eqs2,p21(…),p22(…),…,p2k2(…).
… ,
p(…):-
     eqsn,pn1(…),pn2(…),…,pnkn(…).
```

In the above, each clause corresponds to one case, and the order of the clauses represents the order of cases of the definition `p` set by the programmer in the editor environment. In the ith clause, the head `p(…)` corresponds to the head of the ith case; `eqsi` is the list of equalities corresponding to the function cells in the body of the case; pi1(…), …,$pik_i(\ldots)$ correspond to the k_i literal cells in the body of the case, their order obtained from the order of corresponding literal cells in the layered structure of the ith case window.

As an example, consider the translation to Prolog of the definition of Concat shown in Figure 2.

```
concat(X,Y,Y):-
             X=[].
concat(X,Y,Z):-
```

```
X=[H|T],Z=[H|Temp],
concat(T,Y,Temp).
```

The translation described so far provides clauses and queries which will run correctly in Prolog. However, as we mentioned earlier, when executing a query in **Animated** or **Single Step** mode, the details of merge, deletion and replacement rules need to be visualised. In the next section we show how this is achieved.

5.2 Interpreting Lograph Programs

Lograph and Prolog are both based on Horn clauses. However, what distinguishes Lograph from Prolog is how it reveals the details of resolution and unification through its three execution rules.

Once a Lograph program is translated to Prolog, the steps Prolog takes in executing the query must be refined into a series of applications of the three Lograph execution rules. This is essential for the Lograph editor to be able to provide a visualisation of the execution in **Single Step** or **Animated** mode.

The Lograph replacement rule is a simple version of resolution, in which only variable-to-variable substitution is required. Hence replacement can be handled directly by Prolog. Prolog deals with equalities by resolution with the clause `x = x.`, but for equalities we require merge and deletion, which we therefore need to implement.

During the execution of a query, the interpreter must report to the editor environment when an execution rule is applied, identifying the cell or cells involved, and in the case of replacement, the clause which was used. This requires a one-to-one mapping between the Lograph components in the editor and items in the engine. This is accomplished by assigning unique Ids to literal cells and function cells in the Lograph program, and their corresponding literals and equalities in the Prolog program.

The implementation consists of three modules. The *Editor* implements the visual programming environment of Lograph as described above. It includes an automatic layout algorithm for queries and an XML translator for saving and loading programs and queries. The Editor communicates with the *Model* module, which maintains structures representing Lograph programs and queries, translates them into Prolog for execution, and generates Ids. The Model module communicates with the *Interpreter Engine*, a Prolog program which consists of a set of predicates for executing the three Lograph execution rules and communicating with the Model. Editor and Model communicate in order to build and edit Lograph programs and queries. Model and Engine communicate to manage execution.

To describe execution, we consider the following Prolog query, obtained from a Lograph query as described above:

```
eqList,p1(…),…,pm(…).
```

Model generates unique Ids, `id1`,…,`idm`, for the literals in the above, modifying the query as follows.

```
eqList,p1(…,id1),…,pm(…,idm).
```

When replacement is applied to this query, it will be applied to `p1(…)`, replacing it with the body of the first clause for predicate `p1`. Translated directly from Lograph, this clause will have the form

```
p1(…):-
    eqs1,p11(…),p12(…),…,p1k(…).
```

However, literals in the query have been augmented with Ids, so the head of the clause must be modified, by adding a new variable to match the Id in the query literal. Also, when this replacement is performed, Engine should tell Model that clause 1 for `p1` was used, and pass the identity of the replaced literal. In return, Model should generate Ids

for the new goal literals introduced. This is achieved by a new literal added at the beginning of the body of the clause, and adding new variables to the body literals to accept the Ids generated by Model. With these modifications, the clause has the following form.

```
p1(…,Id):-
        replace(Id,1,[Id1,Id2,…,Idk]),
        eqs1,
        p11(…,Id1),
        p12(…,Id2),
        … ,
        p1k(…,Idk).
```

When the literal `p1(…,id)` in the query is replaced using this clause, the variable `Id` is instantiated to the constant `id`.

The `replace` predicate implements the required interface with Model. The list `[Id1,Id2,…,Idk]` contains new Ids generated by Model for the literals in the body of the substituted clause.

In general, the ith clause for corresponding to a Lograph definition `p` will have the form:

```
p(…,Id):-
        replace(Id,i,[Id1,Id2,…,Idj]),
        eqsi,
        p11(…,Id1),
        p12(…,Id2),
        … ,
        p1j(…,Idj).
```

Next we consider the merge and deletion rules. Since these two rules operate on function cells (equalities in the equivalent Prolog clauses), they also need to have Ids so that when Engine reports the application of merge and deletion rules to Model, it can also report the cells involved. To account for these, the structure of a clause is further expanded, as follows:

```
p(…,Id):-
    replace(Id,i,[EqsIds,[Id1,…,Idj]]),
    matchIds(eqsi,EqsIds,EqsMatched),
    pi1(…,Id1),
    pi2(…,Id2),
    … ,
    pij(…,Idj).
```

Now let us recap the replacement rule, describing the functionality added in the last step. When a replacement occurs, the Id of the replaced literal is already instantiated to a constant. The first literal in the body of the replacing clause, `replace`, will report to Model the Id of the replaced literal together with the case number of the clause. Model uses these to identify which literal cell in its query structure is being replaced. The case number will be used to correctly identify the case in the Lograph program, the body of which will replace the literal cell. This enables Model to generate Ids for the function and literal cells in the body of the replacing case. These Ids are then reported to both Editor and Prolog. Editor will use the Ids to identify the components that need to be redrawn. A successful execution of `replace` also instantiates `[Id1,…,Idj]` and `EqsIds` which are then assigned to the equalities in `eqsi` by execution of the `matchIds` literal.

The next modification deals with the merge rule. Merge is applied to function cells with connected roots and identical names and arities. In the Prolog representation of Lograph queries, merge is applied to equalities which have the form $X = f(X_1,X_2,\ldots,X_s)$ and $X=f(Y_1,Y_2,\ldots,Y_s)$. One of the equalities is removed after variables X_j and Y_j are unified for all $1 \le j \le s$, and the Id of the removed equality is reported back to Model. Model responds by an appropriate action depending on the mode in which the program is running.

The merge rule is applied on any two function cells in the query. Therefore, after a replacement, all the equalities in the query must be inspected for application of merge. This implies that in the body of each clause, all the equalities in the query must be available as well as the new equalities introduced by the clause itself. This is achieved by passing the list of equalities from one predicate to the next in the query, as follows:

```
p1(…,eqsList,EqList1,id1),
p2(…,EqList1,EqList2,id2),
… ,
pm(…,EqList(m-1),EqListOut,idm).
```

Note that in the original version of the query, the list of equalities `eqsList` preceded the first literal. It has now been inserted as a parameter to the first literal. The ith literal takes as a parameter `EqList(i-1)`, the list of equalities that exist once all preceding literals have been executed, and returns a modified list `EqListi`. With this in mind, clauses in the program must be similarly modified leading to the following structure.

```
p(…,EqListIn,EqListOut,Id):-
        replace(Id,i,[EqsIds,[Id1,…,Idj]]),
        matchIds(eqsi,EqsIds,EqsMatched),
        append(EqListIn,EqsMatched,Eqs),
        merge(Eqs,EqList0),
        pi1(…,EqList0,EqList1,Id1),
        pi2(…,EqList1,EqList2,Id2),
        … ,
        pij(…,EqList(m-1),EqListOut,Idj).
```

Here, the `append` literal attaches the incoming list of equalities to the list of equalities introduced by the clause itself. The new list, `Eqs`, is then passed to the `merge` literal which implements the merge rule, producing `Eqlist0`, a list of equalities no two of which have the same left hand side (in Lograph terms, no two function cells connected at their roots).

The clauses that define the `merge` predicate also report to Model both successful and failed applications of merge. Recall that in Lograph, a failure occurs when two function cells connected at their root terminals cannot be merged. This can happen when the two function cells either have different names or unequal arities. When a failure is detected during execution of `merge`, the call to `merge` will also fail, causing backtracking. We will discuss this further below.

Deletion removes from the query all function cells the roots of which have no other occurrences. This corresponds in the Prolog query to deleting equalities the left hand sides of which have no other occurrences. Deciding whether or not an equality is deletable, requires determining whether the variable on the left hand side has other occurrences. Prolog, however, does not keep track of the number of occurrences of a variable and therefore is not capable of making this determination.

To deal with this problem, we keep a count of the number of occurrences of each variable in the query. Every variable in our interpreter engine is represented by a pair, of which the first element is the variable and the second is the number of occurrences.

Maintaining variables in this form requires further modifications to the form of query and clauses, since whenever a replacement, merge or deletion takes place, the number of occurrences of the variables involved in the application of the rule need to be updated. This leads us to the following format for the query.

```
p1(…,eqsList,EqList1,varsIn,Vars1,id1),
p2(…,EqList1,EqList2,Vars1,Vars2,id2),
… ,
pm(…,EqList(m-1),EqListOut,Vars(m-1),VarsOut,idm).
```

Here, `varsIn` is the list of variables occurring in the query. When the ith literal is executed, `Vars(i-1)` is the current list of variables in the query. Execution of the literal produces an updated list `Varsi`. Clauses in the program are further modified to include this process, as follows:

```
P(…,VarsIn,VarsOut,EqListIn,EqListOut,Id):-
    replace(Id,i,[EqsIds,[Id1,…,Idj]]),
    unify(VarsIn,updateVars,Vars),
    matchIds(eqsi,EqsIds,EqsMatched),
    append(EqListIn,EqsMatched,Eqs),
    merge(Vars,VarsMerged,Eqs,EqsMerged),
    delete(VarsMerged,Vars0,EqsMerged,EqList0),
    Pi1(…,Vars0,Vars1,EqList0,EqList1,Id1),
    Pi2(…,Vars1,Vars2,EqList1,EqList2,Id2),
    … ,
    Pij(…,Vars(m-1),VarsOut,EqList(m-1),
                                    EqListOut,Idj).
```

In the above, `updateVars` consists of new variables introduced by this clause, together with variables that appear in the head of the clause. The counts the latter need to be adjusted as a result of replacement of a literal in the query using this clause. The second element of each variable pair in `updateVars` is an integer (possibly negative), computed by subtracting the number of occurrences of the variable in the head of the clause from the number of occurrences in the body of the clause. Clearly, if a variable has the same number of occurrences in the head and the body of the clause, it can be omitted from `updateVars`.

As mentioned earlier, our interpreter implements merge and deletion explicitly while relying on Prolog for search, backtracking and the replacement rule. Although in case of failure Prolog will successfully undo the application of Lograph rules, it cannot undo the side effects of predicates which communicate with Model. There are three such predicates, `replace`, and two other predicates that report to Model the application of merge and deletion, along with the Ids of the involved equalities. However, in order for Lograph to provide a visualisation of backtracking, our interpreter engine must also report to Model the undoing of replacement, merge and deletion during backtracking. We will show how the `replace` predicate is modified to accomplish this. The other two predicates which cause side effects can be modified in a similar way.

We introduce another predicate called `undo_replace` which reports to Model the Id of the replaced literal and the number of the clause used in the replacement.

```
P(…,VarsIn,VarsOut,EqListIn,EqListOut,Id):-
        (replace(Id,i,[Eqs_Ids,[Id1,…,Idj]]) |
        undo_replace(Id,i),fail),
        unify(VarsIn,updateVars,Vars),
        …
```

Backtracking into the OR structure causes execution of the `undo_replace` literal which reports the Id and the clause number to Model. This is immediately followed by execution of `fail` which stops execution of the clause body from being repeated.

As an example, consider the two cases of Concat in Figure 2. Model creates the following two clauses which it passes to Engine.

```
concat(X,Y,Y,VarsIn,VarsOut,EqListIn,EqListOut,Id):-
        (replace(Id,1,[EqListIn,[]]) |
                            undo_replace(Id,1),fail),
        unify(varsIn,[v(Y,-2)],Vars),
        matchIds([e(X=[])],EqsIds,EqsMatched),
        append(EqsListIn,EqMatched,Eqs),
        merge(Vars,VarsMerged,Eqs,EqsMerged),
        delete(VarsMerged,VarsOut,
                                EqsMerged,EqListOut).
concat(X,Y,Z,VarsIn,VarsOut,EqListIn,EqListOut,Id):-
        (replace(Id,2,[EqsIds,[Id1]]) |
                        undo_replace(Id,2),fail),
        unify(varsIn,[v(H,2),v(T,2),v(Temp,2)],Vars),
        matchIds([e(X=[H|T]),e(Z=[H|Temp])],
                                EqsIds,EqsMatched),
        append(EqsListIn,EqsMatched,Eqs),
        merge(Vars,VarsMerged,Eqs,EqsMerged),
        delete(VarsMerged,VarsOut,EqsMerged,EqList0),
        concat(T,Y,Temp,Vars0,VarsOut,
                                EqList0,EqOut,Id1).
```

A more detailed description of the engine including the implementation of `replace`, `undo-replace`, `merge`, and `delete` along with the implementation of the interface to Model can be found in [2].

6 Summary

We have identified problems that arise when using Prolog for interpreting Lograph programs and proposed solutions for them. Prolog translations of Lograph programs are probed at appropriate places so that applications of Lograph rules can be reflected in the visual representation of a program. Our interpreter takes advantage of the search and backtracking mechanisms of Prolog while implementing the delete and merge rule exclusively.

We have also briefly discussed the Lograph programming environment, including the visualisation of the execution order of literals in cases, and animation of execution.

Our future work on Lograph will focus on improving the automatic layout algorithm to better preserve the user's mental map of the query under execution and our ultimate goal is to extend Lograph to obtain LSD.

It is agreed that a visual logic programming language would be beneficial for teaching logic programming [5,11]. We believe that Lograph can also be a valuable programming environment for teaching logic programming as well as a useful editing and debugging environment for logic programs. Animating the transformation of a query and backtracking can be a valuable substitute for typical textual tracing techniques. This, however, cannot be supported without empirical studies which have yet to be performed.

7 References

[1] Adobe Systems Inc., *Photoshop 6.0 User Guide*, (2000).

[2] O. Banyasad, P. T. Cox, *Interpreting Lograph*, Dalhousie University, CS-2003-03.

[3] P.T. Cox, T. Pietrzykowski, LOGRAPH: a graphical logic programming language, *Proceedings IEEE COMPINT 85*, Montreal (1985), pp 145-151.

[4] P.T. Cox, T. Smedley, LSD: A Logic Based Visual Language for Designing Structured Objects, *Journal of Visual Languages and Computing*, v9, Academic Press (1998), 509-534.

[5] M. Eisenstadt, M. Brayshaw, A fine-grained account of Prolog execution for teaching and debugging, *Instructional Science*, 19(4/5), pp.407-436, 1990.

[6] M. Eisenstadt, M. Brayshaw, The Transparent Prolog Machine (TPM): An execution model and graphical debugger for logic programming, *Journal of Logic Programming*, 5(4), pp 277-342 (1988).

[7] K.M. Kahn, V.A. Saraswat, Complete Visualizations of Concurrent Programs and their Executions, *Proceedings of the IEEE Workshop on Visual Languages*, (1990), pp 7-15.

[8] E. Knill, P.T. Cox, T. Pietrzykowski, Equality and abductive residua for Horn clauses, *Theoretical Computer Science*, 120 (1993), pp 1-44.

[9] D. Ladret, M. Rueher, VLP: a Visual Programming Language, *Journal of Visual Languages and Computing*, 2(2), pp 163-189 (1991).

[10] M. A. Najork, Programming in Three Dimensions, PhD Thesis, University of Illinois at Urbana -Champaign, Urbana, Illinois, 1994.

[11] E. Neufeld, A. J. Kusalik, M. Dobrohoczki, Visual metaphor for Understanding Logic Program Execution, *Proceedings of Graphics Interface*, pp 114-120, 1997.

[12] L.F. Pau, H. Olason, Visual Logic Programming, *Journal of Visual Languages and Computing*, v2 (1991), pp 3-15.

[13] J. Puigsegur, W.M. Schorlemmer, J. Agustí, From Queries to Answers in Visual Logic Programming, *Proceedings of the IEEE Symposium on Visual Languages*, (1997), pp 102-109.

[14] H. Senay, S. G. Lazzeri, Graphical representation of logic programs and their behaviour, *Proceedings of the 1991 Workshop on Visual Languages*, pp 25-31.

[15] Swedish Institute of Computer Science, *SICStus Prolog User's Manual*, release 3.10.0 (2002).

[16] L.L. Spratt, A.L. Ambler, A Visual Logic Programming Language Based on Sets and Partitioning Constraints, *Proceedings of the IEEE Symposium on Visual Languages* (1993), pp 204-208.