



Interpreting Lograph

Omid Banyasad
Philip T. Cox

Technical Report CS-2003-03

Mar 21, 2003

Faculty of Computer Science
6050 University Ave., Halifax, Nova Scotia, B3H 1W5, Canada

Interpreting A Visual Logic Programming Language

Omid Banyasad

Philip T. Cox

Faculty of Computer Science
Dalhousie University, Halifax, Nova Scotia, Canada
(+1-902) 494-6460

banyasad@cs.dal.ca

pcox@cs.dal.ca

ABSTRACT

Lograph is a non-deterministic, visual, logic programming language which is being implemented as the basis for a visual language for the design of structured objects.

Given the multi-paradigm nature of the language, implementing it raises many issues such as a suitable editing and debugging environment, a deterministic execution mechanism, and an efficient interpreter engine.

The consideration of deterministic execution raises another issue. In logic programming languages, deterministic execution is obtained by ordering literals in the body of clauses, and ordering clauses in the definition of a predicate, how should such ordering be expressed in a visual logic programming language without resorting to a confusing network of lines?

Lograph's roots in logic raises the issue of efficient execution as it is the case for any other logic programming language. Is it possible to take advantage of the considerable effort that has gone into logic program compilation?

This paper describes our progress in addressing such issues. We describe how Lograph can be made deterministic and potentially efficient by appropriate orderings, and how these orderings can be represented and managed. We then discuss the design of an interpreter engine which takes full advantage of an efficient implementation of Prolog while providing for animation of execution.

1 INTRODUCTION

Language for Structured Design (LSD), is a visual language for designing structured objects [7]. 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 [5]. Hence, in order to implement LSD we must first implement Lograph.

Although various visual logic programming languages have been proposed, for example [10,11,13], 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 [6]. 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 design language, where animating the assembly of objects is a desirable feature.

In this paper, we report on our progress towards implementing an industrial strength Lograph as a general purpose logic programming language, to be later used as the basis for LSD.

2 LOGRAPH SYNTAX AND SEMANTICS

Our presentation in this section is based on [5,6] 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. 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 $\frac{\langle \text{name} \rangle}{\circ}$ or $\frac{\circ}{\langle \text{name} \rangle}$, where $\langle \text{name} \rangle$ is the name of the function.



Figure 1: Function cells

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. 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 $\frac{\langle \text{name} \rangle}{\circ}$ or $\frac{\circ}{\langle \text{name} \rangle}$, where $\langle \text{name} \rangle$ is the name of the function.

In Lograph, as in other logic programming languages, function cells can be used to create data structures. For example, an empty 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 2 the constant $\frac{\circ}{\square}$ represents the empty list, and function cells named \bullet 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 2 can be denoted $\overline{[1,2,1,2]}$.

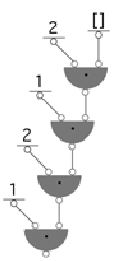


Figure 2: A list

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 3 shows a literal cell named Concat

Figure 3: A literal cell



with arity 3.

Figure 4 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 4 represent the two flat Horn clauses:

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

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

The semantics of flat Horn clauses are provided by *Surface Deductions* set of three deduction rules discussed in detail in [6, 9]. 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.

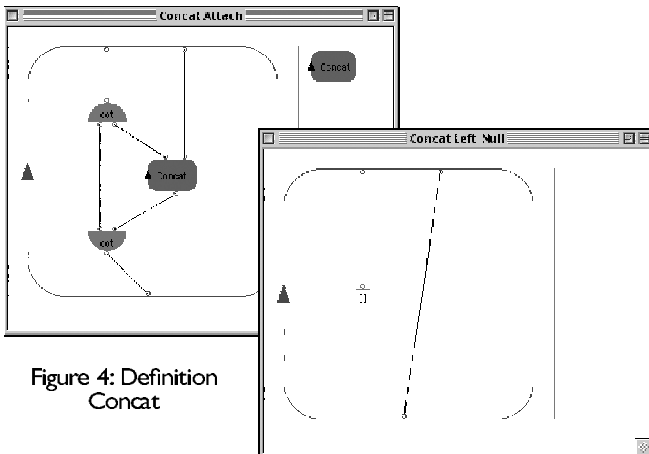


Figure 4: Definition Concat

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.

2.3 Lograph and Prolog

In Section 3, we present the restrictions we impose on Lograph to obtain a viable programming language. 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, and 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. It is important to remember, however, that our purpose in implementing Lograph is to provide a basis for LSD where the goal is to generate geometric solids in a design space. In that context, the graphical transformations accomplished by the merge and deletion rules (finer grained than unification in Prolog) are important; and the definition of “successful computation” depends to a large extent on the nature of the solids produced in the design space.

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 DETERMINISTIC LOGRAPH

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. Because Lograph represents flat Horn Clauses graphically, it expresses this non-determinism in a natural way: however, 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 that 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.

3.1 Order of Transformation Rules

In this section we show 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.

As mentioned in section 2.2, the deletion rule is applied to functions with dangling root terminals. Since a *deletable* function cell cannot participate in any other transformation rules, the order in which deletable cells are removed will have no effect on the rest of the execution.

Let us suppose that the replacement rule precedes the merge rule in the chosen rule ordering, and that the current query contains a pair {A, B} of function cells connected by their roots. Clearly A and B are compatible, since otherwise the query cannot be reduced to the empty graph.

We have four cases to consider: either

- (a) the query contains some literal cells, *or*
the query does not contain any literal cells, and, *either*
- (b) the roots of the two compatible cells identified above are not connected to any other terminals, *or*
- (c) the roots of the two cells are connected to the roots of some other function cells, *or*
- (d) the roots of the two cells are connected only to non-root terminals of some other function cells.

In case (b), the only transformations that can be applied to A and B is a merge followed by a deletion. These transformations are independent of any others, and can therefore be applied immediately.

In case (c), since the query is eventually reduced to the empty graph, any function cell connected by its root to the roots of A and B must be compatible with them. Since every merge produces a function cell compatible with the merged cells, and therefore with any other compatible cells attached to them by root terminals, the order in which the cells in such a group are merged is irrelevant. Hence the merge of A and B can be performed before any other merges of cells in the group.

In case (d), suppose that the execution removes the “other” function cells before any other transformations occur. The “other” cells are removed either by (d1) deletion, or by (d2) merging followed by

deletion.

In case (d1), we are left with an instance of case (b) or (c), so that merging A and B can be the next operation performed. Clearly the deletion of the “other” function cells does not depend on the presence of cells A or B, so the merging of A and B could be performed earlier.

In case (d2), we are left with an instance of case (b), (c) or (d). We deal with (b) and (c) as in the previous paragraph. As for (d), we need only note that cases (d) and (d2) cannot alternate forever since the transformations that occur in case (d2) strictly reduce the size of the graph, so we must eventually get case (b) or (c).

In case (a), no merging of A and B will occur until all replacements have been performed. Clearly, performing the replacements is not affected by the presence or absence of A or B, so we can merge A and B at any time.

Since, as we have seen, the transformation rules can be applied in any order, we need to consider the best order in which to apply them. Obviously, if we are doing a depth first search of the solution space as in Prolog, then we should try to discover “nonunifiability” early. In Lograph, this means applying the merge rule as early as possible. The deletion rule does not really affect this since it just plays the role of “garbage collector”; however, if we are interested in useful animated visualisations of executions, then we might want to apply it early as well, in order to reduce clutter.

Another issue relating to the order of rule application is whether or not we want to optimise our search for a solution. Interestingly, the graphical structures built from function cells are very similar to structures for terms proposed in [3] to enable intelligent backtracking. If intelligent backtracking were to be implemented, the merge and deletion rules would not necessarily be applied as early as possible.

Finally, since Lograph replaces unification with explicit transformation rules, there may be ways to apply them which are better suited to the application. For example it might be possible in some circumstances to “batch” merges and deletions, applying them only occasionally between sequences of consecutive replacements.

3.2 Ordering Cases and Cells


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. Specifying the order, however, presents an interesting problem. Like other visual languages, Lograph exposes the structure of algorithms without imposing needless sequentiality; nevertheless, sequentiality is required for the sake of efficiency. This looks like the same problem that arises in implementing other visual languages, for example dataflow languages. In the case of dataflow languages, operations are partially ordered, and any linear order produced by topological sort will do [4]. In Lograph, however, the wires are not data flow links, so no suitable order can be automatically generated. It is therefore up to the Lograph programmer to specify an order that

will do. Prolog programmers order literals by textually writing them in a linear order. This cannot be done with Lograph cells.

One obvious solution is to add special connections between literal cells to indicate execution order, like the synchros of Prograph [4]. However these would be far more intrusive than in Prograph where data flow provides most of the ordering so synchros are needed only occasionally. The solutions we have adopted in the Lograph environment are described in the next section.

4 PROGRAMMING ENVIRONMENT

When Lograph is started, a menu bar containing, **File**, **Edit**, **Run**, and **Settings** appears together with an empty window named **Untitled** in which a program is created and maintained. Double-clicking in this window creates a definition icon with no terminal, an origin on its left end, and the default name Un-named. Every definition icon has a sensitive boundary; that is, the cursor changes to  whenever it is near the perimeter of the cell, indicating that a click will add a terminal.

Double-clicking on the definition opens its *cases* window, consisting of the list of cases on the right and a thumbnail of the selected case on the left. A new case is added to the definition of a literal by double clicking in the list. This creates a new case named Case N where N is the number of cases previously created. The name of a case can be edited at any time by selecting it in the list and typing. Note that our implementation allows cases to be named, which is not a feature of Lograph as described above. The order of cases can be changed by dragging them in the list. This order is the order in which cases will be applied during execution.

Double clicking on a case in the case list or on the thumbnail of the selected case, opens a *case* window consisting of a *workspace* to the left and a *layer list* to the right, both of them initially empty. Figure 4 illustrates two case windows. The workspace and layer list provide two representations of literal-cell ordering, 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. The transparency of icon cells is user-adjustable. 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.

A new literal cell can be added to the workspace by dragging and dropping a definition from the program window or by double clicking in the workspace. Definition icons in the program window and literal cell icons in case windows are similar except for their colours, green and blue respectively. The colour settings for different icon classes can be customized. Figure 4 illustrates two case definition windows of a literal named **Concat**.

A new function cell can be added to the workspace by double clicking while holding down the "F" key. The new function cell has arity 0 and named Un-named by default which can be edited. Terminals can be added by clicking at the sensitive boundary of the cell around its flat face. Double clicking a function cell changes its orientation, from pointing-up to pointing-down and *vice versa*. All the function cells in the workspace are automatically placed on the top-most layer.

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 considering that the interpreter itself was interpreted. 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 able to support the demands of the intended application.

Our initial Lograph prototype provided capabilities similar to pure Lisp and Prolog. However, in order to make Lograph suitable for our intended application, some of the extra features of those languages need to be included as well. For example, although logic programming is not usually used for numerical problems, Prolog provides numbers as special constants and some basic arithmetic operations that compute functions of numbers in a data flow fash-

ion. In the domain of structured object design, there is a clear need for numerical computation far more extensive than that normally expected in pure logic programs, so it will be necessary to extend Lograph to provide this capability beyond the mechanisms provided by pure logic programming languages, which are inconvenient at best.

Based on the above observations, we chose the SICStus implementation of Prolog [12] as the basis for Lograph's interpreter. The latest release of SICStus Prolog 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. Consequently, 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 in the form of $x = f(\dots)$ and a list of flat literals $p_1(\dots), p_2(\dots), p_3(\dots), \dots, p_m(\dots)$, where each \dots 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 $p_1(\dots)$ to $p_m(\dots)$ 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 in the cases window of the definition *p*. In the i^{th} clause, the head $p(\dots)$ corresponds to the head of the i^{th} case; *eqs_i* is the list of equalities corresponding to the function cells in the body of the case; $p_{i1}(\dots), \dots, p_{ik_i}(\dots)$ 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 i^{th} case window.

As an example, consider the translation to Prolog of the definition of *Concat* shown in Figure 4.

```
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 Lograph visually 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 needs to be translated to 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 a **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 $x = x \text{ :-}$, but for equalities we require merge and deletion, which we therefore need to implement.

During the execution of a Lograph 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 corresponding Prolog.

Figure 5 shows a high-level block diagram of the front and back ends of the Lograph implementation.

In Figure 5, *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. *Model* is a module that maintains structures representing Lograph programs and queries, translates

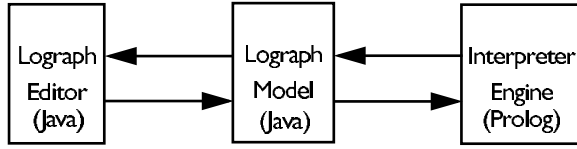


Figure 5: Implementation Block Diagram

them into Prolog for execution, and generates Ids. The *Interpreter Engine* is 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, id_1, \dots, id_m , 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),
    ... ,
    p1k1(..., Idk).
```

When the literal $p1(..., id1)$ in the query is replaced using this clause, the variable Id is instantiated to the constant $id1$.

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

In general, the i^{th} clause for corresponding to a Lograph definition p will have the form:

```
p(..., Id) :-
    replace(Id, i, [Id1, Id2, ..., Idj]),
    eqs1,
    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(eqs1, EqsIds, EqsMatched),
    p11(..., Id1),
    p12(..., Id2),
    ... ,
    p1j(..., 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 $[Id1, \dots, Idj]$ and `EqsIds` which are then assigned to the equalities in `eqs1` 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 $X = f(X_1, X_2, \dots, X_s)$ and $X = f(Y_1, Y_2, \dots, Y_s)$. One of the equalities is removed after variables X_j and Y_j are unified for all $1 \leq j \leq s$, 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 i^{th} 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 i^{th} 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,[EqsIds,[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 4. Model creates the following two clauses which it passes to Engine.

```

concat(X,Y,Y,VarsIn,VarsOut,
    EqListIn,EqListOut,Id):-
    (replace(Id,cn,[EqListIn,[]]) |
     undo_replace(Id,1),fail),
    unify(varsIn,[v(Y,-2)],Vars),
    matchIds([e(X=[])],EqsIds,EqsMatched),
    append(EqsListIn,EqsMatched,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 reported on our progress towards implementing Lograph, a visual, logic programming language intended as the basis for a visual language for design of structured objects.

For execution efficiency, it is necessary to restrict the Lograph language from a general first-order Horn-clause theorem-prover to an efficiently implementable language. Many of the restrictions are obvious counterparts of the restrictions inherent in Prolog, involving the ordering of clauses and the ordering of literals.

The front-end of the language is a visual programming environment for creating, editing, and debugging Lograph programs implemented in Java. The core of the language, also in Java, implements the semantics of Lograph with proper interfaces to the front-end as well as an interpreter engine implemented in Prolog.

Since Lograph is based on surface deduction rather than simple resolution, it is also necessary to consider how its three execution rules should be ordered. We have shown that the merge and deletion rules can be applied at any time, and to simulate the search order of Prolog, should be applied as early as possible.

Like other visual languages, Lograph exposes the structure of algorithms in a useful way. However, since it is not dataflow, there is no way to automatically linearise operations, so like the Prolog programmer, the Lograph programmer must take responsibility for this task. We have proposed and implemented a layering scheme similar to the layering of Photoshop images, for visualising the ordering of literal cells in a case.

We have also identified issues arisen in using Prolog for interpreting Lograph programs and proposed solutions for them. Prolog translation of Lograph programs are probed at appropriate places such that equivalent applications of Lograph rules are reported to Lograph as the side effect of the execution of a program. Our interpreter takes advantage of search mechanism and backtracking of

Prolog while implements the delete and merge rule exclusively. Our interpreter in Prolog is currently implemented and fully functional.

We believe that Lograph can 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.

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, On determining the causes of nonunifiability, *Journal of Logic Programming* 4, American Elsevier (1987), pp 33-58.
- [4] P.T. Cox, F.R. Giles, T. Pietrzykowski, Prograph: a step towards liberating programming from textual conditioning, *Proc. 1989 IEEE Workshop on Visual Programming*, Rome (Oct. 1989), 150-156. Reprinted in *Visual Object-Oriented Programming: Concepts and Environments*, M. Burnett, A. Goldberg, & T.G. Lewis (Eds), Manning Publications (1995).
- [5] P.T. Cox, T. Pietrzykowski, LOGRAPH: a graphical logic programming language, *Proceedings IEEE COMPINT 85*, Montreal (1985), pp 145-151.
- [6] P.T. Cox, T. Pietrzykowski, Incorporating equality into logic programming via Surface Deduction, *Annals of Pure and Applied Logic* 31, North Holland (1986), pp 177-189.
- [7] 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.
- [8] K.M. Kahn, V.A. Saraswat, Complete Visualizations of Concurrent Programs and their Executions, *Proc. IEEE Workshop on Visual Languages*, (1990), pp 7-15.
- [9] E. Knill, P.T. Cox, T. Pietrzykowski, Equality and abductive residua for Horn clauses, *Theoretical Computer Science*, 120 (1993), pp 1-44.
- [10] L.F. Pau, H. Olason, Visual Logic Programming, *Journal of Visual Languages and Computing*, v2 (1991), pp 3-15.
- [11] J. Puigsegur, W.M. Schorlemmer, J. Agustí, From Queries to Answers in Visual Logic Programming, *Proc. IEEE Symposium on Visual Languages*, (1997), pp 102-109.
- [12] Swedish Institute of Computer Science (2002), SICStus Prolog User's Manual release 3.10.0.
- [13] L.L. Spratt, A.L. Ambler, A Visual Logic Programming Language Based on Sets and Partitioning Constraints, *Proc. 1993 IEEE Symposium on Visual Languages*, (1993), pp 204-208.