From Queries to Answers in Visual Logic Programming*

Jordi Puigsegur [1,†], W. Marco Schorlemmer [1,2,‡] and Jaume Agustí [1]

[1]Institut d'Investigació en Intel·ligència Artificial
Campus UAB, 08193 Bellaterra, Catalonia, EU
{jpf,marco,agusti}@iiia.csic.es

[2]SRI International, Computer Science Lab
333 Ravenswood Ave, Menlo Park, CA 94025, USA
marco@csl.sri.com

Abstract

In VL'96 we presented a visual declarative programming language based on two main graphical constructs: directed acyclic graphs representing predicate application and graphical set inclusion representing logical implication. We showed that with these simple visual constructs we can cover most of the representational demands of computational logic allowing a blend of functional and relational styles of programming. In this paper we explore the advantages of using directly our visual syntax for solving queries, by giving a way to visually ask questions about a visual program by means of query diagrams, and by defining visual inferences which operate on those diagrams. The result is an operational semantics for declarative programming which is intended to be visual, intuitive and formal. Visual because the inference rules display graphically the transformation of query diagrams into answer diagrams. Intuitive because it is intimately linked with the visual syntax of the declarative language. Finally it is formal because the usual properties of formal logic (i.e. correctness and completeness) can be applied directly to it.

1. Introduction

In VL'96 [12] we presented a visual declarative programming language based on two main graphical constructs: directed acyclic graphs representing predicate application and graphical set inclusion representing logical implication. We showed that with these simple visual constructs we can cover most of the representational demands of computational logic allowing a blend of functional and relational styles of programming. However graphical inclusion not only has an intuitive representational power but visualizes the basic kind of reasoning used in computational logic. To show this is the goal of this paper.

The operational semantics is a key point in the definition of a declarative programming language. In order to use the declarative language as a programming tool we need to know how it behaves during execution time, and this is what operational semantics is for: to specify the operational behavior of the language. The success of a declarative programming language depends mainly on the simplicity and the intuitiveness of the inference rules that define its operational semantics. For instance, Prolog's success may be attributed to the simplicity of resolution, that allows programmers to have a mental image of the execution of their program while they are writing it. Back to the visual ground, we think that the operational semantics of a visual declarative programming language has to be:

Visual: defined mainly using visual inference rules that show how diagrams are transformed.

Intuitive: intimately linked with the syntax of the language. From these visual inference rules we have to be able to see how the diagrams are going to be executed.

Completely formal: we have to be able to prove the usual properties (soundness, completeness, etc.) of these visual inference rules.

In the diagrammatic reasoning field there exist many completely visual formal reasoning systems [2]. VEX [4], a visual notation for λ-calculus, introduces visual transformation rules parallel to the λ-calculus textual ones. However, most of the existing visual declarative programming languages do not have a fully visual operational semantics. CUBE [9, 10] has visual queries which may be visually solved, but relies on an equivalent textual language in order to formally define the operational semantics of the language. Other languages like VLP [6] or VPP [11] are built as front-ends of a textual language, i.e. Prolog. The main contribution of our paper is to present our first attempt at constructing a fully visual operational semantics for our logic programming language [12, 1] which is completely formal and independent of any textual translation of the diagrams.

2. The Language

In this Section we will briefly present the visual declarative language on which this article is based —a fragment of

*Research supported by project DISCOR (TIC 94-0847-C02-01) funded by the CICYT

†Supported by a doctoral grant from the *Direcció General de Recerca* (Catalan Head Office for Research)

‡On leave supported by a grant from the *Direcció General de Recerca*

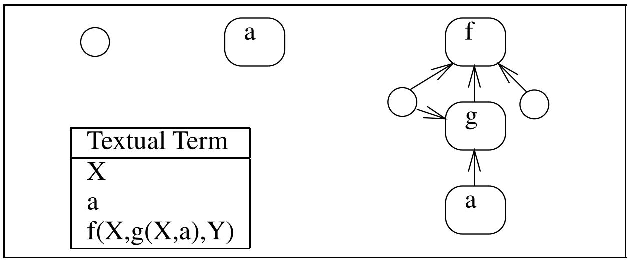


Figure 1. Examples of Terms

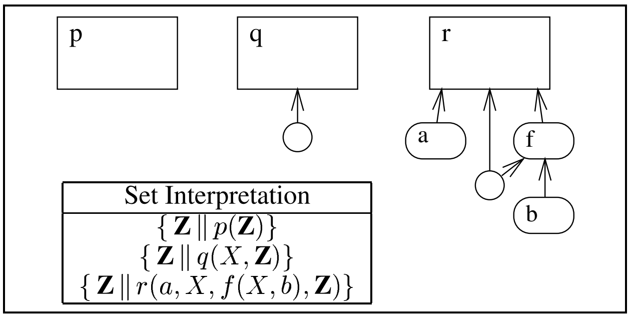


Figure 2. Examples of Predicates

that presented in VL'96 [12]. The visual syntax of our language is a customization of higraphs, an open-ended topological formalism developed by David Harel [5] that combines two well-known diagrammatic formalisms: graphs and Venn diagrams. A higraph is a graph where its nodes —which represent sets and elements— are related by a binary relation: graphical inclusion. Actually there are two important differences with Venn diagrams: first, the graphical sets of our diagrams do not have a unique physical space, i.e. a set may be repeated in various places of the diagram; and second, only graphical inclusion information is relevant, the absence of graphical inclusion is not taken into account. For instance, sets that do not graphically overlap might still have common elements.

In our language we have two different types of graphical items: sets and elements. Arrows are used to represent the application of predicates and functions to terms, as shown below. We combine these two types of graphical items: first by means of functional applications and, second, by graphical inclusion. A diagram defines a predicate by giving subsets and/or elements of the graphical set associated to the predicate. Diagrams can thus be viewed as conditional subset or membership statements, where the conditions are the rest of membership relations of the diagram. We will focus on the exact construction and meaning of the diagrams later in Section 2.3. Let us first see how terms and predicates are visually presented:

2.1. Terms (Elements)

Terms are directed acyclic graphs built using circles and labeled rounded boxes. They represent the elements and are used as predicate arguments. Rounded boxes are those parts of the terms with an explicit name: constants and functions. Variables are presented as circles, which in our notation do not need to have a name. Compound terms are built as graphs by applying functions to other terms using arrows. In Figure 1 we can see examples of how terms are visually presented.

2.2. Predicates (Sets)

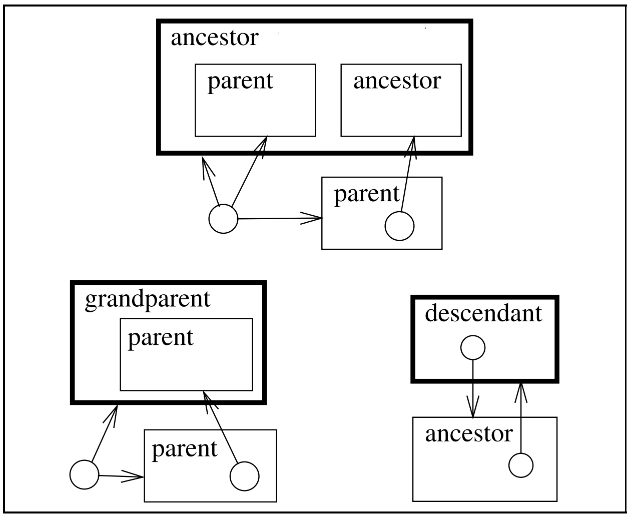
Predicates are visually presented by labeled boxes, meaning the set comprehension over one of its arguments. We assume therefore that one of the predicate arguments is left free and is used as the range for the set comprehension. The rest of the arguments of the predicate are explicitly shown by means of arrows. If we refer to the selected argument —by convention usually the last one— as the variable **Z**, then the set consists of the elements such that instantiating **Z** with them, the predicate holds. In Figure 2 we find three examples of predicates, their visual presentation and the formal interpretation as sets. In this way, predicates are viewed as non-deterministic functions: one argument is chosen as output and the rest are inputs.

2.3. Diagrams

Once we know how to present terms and predicates using the visual syntax, the next step is to study how we can combine different predicates and terms in order to express the intended meaning. A diagram, the visual equivalent of a formula in a textual language, is the smallest complete unit of description in our language.

A diagram is composed of various predicates and terms put together in a single higraph and sharing common subterms. Each diagram has a special predicate —marked using thick lines to draw its box— which is the predicate we are defining, and which in principle only may have at most one graphical item included in it, either a rounded box or circle, or another predicate box. But in practice more than one graphical item may be included within it, expressing different alternatives. Actually this can be seen as a 'shorthand' for presenting in one unique diagram several separate ones defining the same predicate. Somehow, this predicate is equivalent to the head predicate of a clause in Prolog. In fact, diagrams are conditional Horn rules with subset and membership facts as conclusion and several memberships as condition. This is an important restriction: while we can define the predicate using both membership and subset relationships, the conditions, i.e. the rest of inclusions on the diagram are restricted to be only memberships (inclusions of elements into sets).

In Figure 3 we find some examples of diagrams defining the predicates *grandparent*, *ancestor* and *descendant* assuming that a predicate *parent* is already defined (refer to Figure 5 for an example of definition of the *parent* relation). In Figure 3 we can see how these predicates are



$parent(X) \subseteq ancestor(X)$
$ancestor(Y) \subseteq ancestor(X) \Leftarrow Y \in parent(X)$

$parent(Y) \subseteq grandparent(X) \Leftarrow Y \in parent(X)$

$Y \in descendant(X) \Leftarrow X \in ancestor(Y)$

Figure 3. Examples of Diagrams

defined via the subset and membership relations with conditions expressed as membership relations. From their textual translations we see that a visual declarative program is a Horn logic theory with two predicates: $\subseteq$ and $\in$.

Let us now very briefly explain how the semantics of this diagrammatic language is defined. We already said that the semantics of a term is an element of the semantic domain, and that the semantics of a predicate is its set comprehension over one of its arguments. Therefore a model of a diagram will be an interpretation such that, whenever all the memberships in the thin-line boxes hold, the graphical inclusion (membership or subset) within the thick-line box holds. In the next Section we explain how we have defined an operational semantics for this visual language.

3. Visual Query Solving

In previous stages of the development of our visual language we put our efforts on defining its visual syntax [12], while queries were solved translating our diagrammatic programs into pure Prolog. But in order to have a real visual logic programming language we need to explore the advantages of using directly our visual syntax for solving queries, by giving a way to visually ask questions about a visual program by means of *query diagrams*, and by defining *visual inferences* which operate on those diagrams (i.e. a *visual operational semantics* for our diagrammatic language).

As mentioned before, our syntax is based on graphical inclusion between graphical items. In fact, visual programs are actually logical theories involving inclusions, either subset relations or memberships. Levy and Agustí noticed that

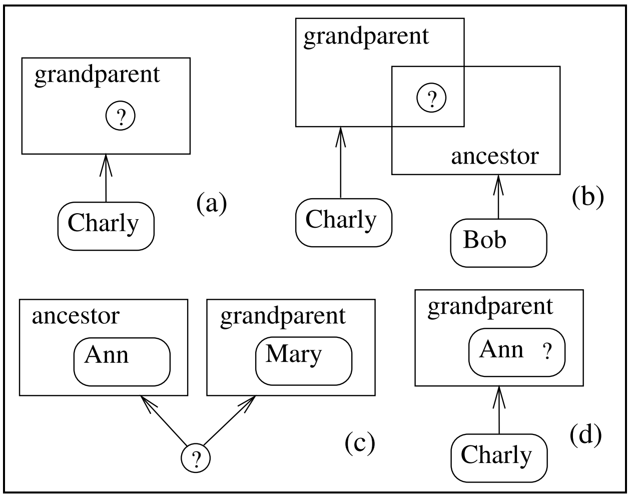


Figure 4. Query Diagrams

rewrite techniques can be applied to the deduction with inclusions in a similar way they have been used for the deduction with equality relations [7, 8], and Bachmair and Ganzinger generalized these techniques by means of their *ordered chaining calculus* within the context of resolution-based theorem proving [3]. Their work serves us as proof-theoretic basis underlying the operational behavior of the *visual query solving* in our language.

3.1. From Queries to Answers

Queries are solved by transforming *query diagrams* into *answer diagrams* applying visual inferences. The idea behind this diagram transformation is to complete the query with the trace of its proof, while instantiating its unknown variables. As we will see later in Section 3.3, queries are solved actually by refuting its negation (like in Prolog).

A *query diagram*, instead of defining a predicate, expresses an existential fact to be proved. In Figure 4 we find different examples of query diagrams related to the visual program of Figure 5. Suppose we want to know who is grandparent of Charly. We express this query with Diagram (a) of Figure 4, in which the variable marked with an interrogation mark is our *goal term.* The textual translation of this query diagram would be:

$$\exists X \; X \in grandparent(Charly) \; ?$$

Later on, in Section 3.2, we will define exactly how query diagrams are constructed, and explore the rest of examples of Figure 4.

As in conventional logic programming, we solve a query by exploring the definitions given in the program. The transformation of the query diagram is then done in two different ways: On one hand we may match parts of the query diagram with definitions given in our visual program, on the other hand we can add into the query *included* boxes present in our definitions.

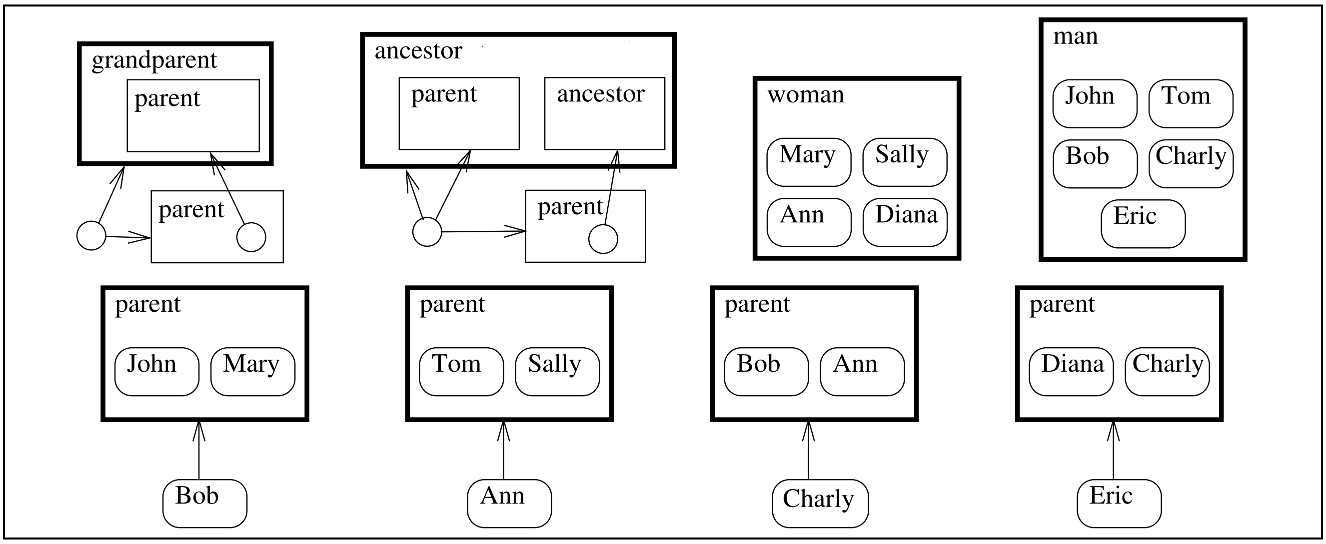
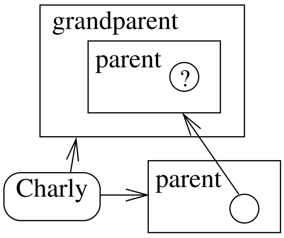


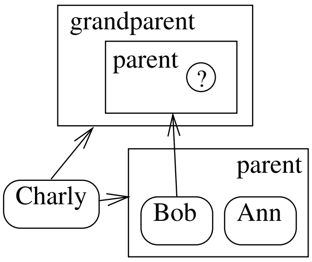
Figure 5. A Visual Program

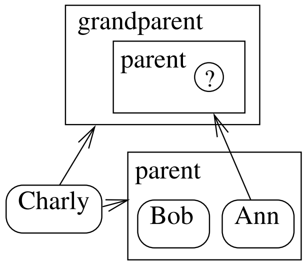
This will become clear by looking at an example. Let's solve the simple query expressed by Diagram (a) of Figure 4. By matching the $grandparent$ box with its definition in Figure 5, we transform the query diagram by replacing the circle —the argument of grandparent— by round box $Charly$, and by adding the included boxes of the definition referring $parent$ to the query diagram:



This inference step captures the idea that we are going to proof the existence of an element in the $grandparent$ box by means of its membership in the $parent$ box contained in it. Therefore we place the unknown variable within the $parent$ box. The transitivity of the subset relation with respect to set membership, is clearly visualized by the diagram, and so we visualize a basic kind of computational logic reasoning. Note that the new query diagram reflects both the new inferred membership of the goal variable into the $parent$ box and the old one into the $grandparent$ box. In this way we keep track of the original query during subsequent diagram transformations.

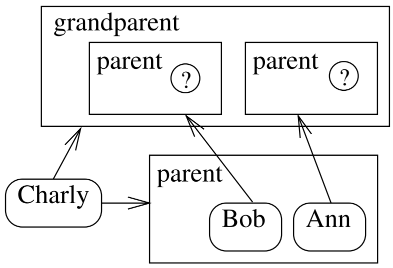
Further diagram transformations involve steps in which no new boxes are added to the diagram, but a box is matched with its definition on the visual program. At this point we have two different possible steps of this type, both of them involving the two *parent* boxes somehow related to the goal term. We have chosen the box with all arguments instantiated, i.e the $parent$ box with argument *Charly*, which is matched with its definition in Figure 5. This gives us two alternatives to instantiate the circle within $parent$, either by round box Bob or Ann:





We do not add —unlike the previous step— new predicate boxes to the diagram; we only instantiate some of the circles due to matching.

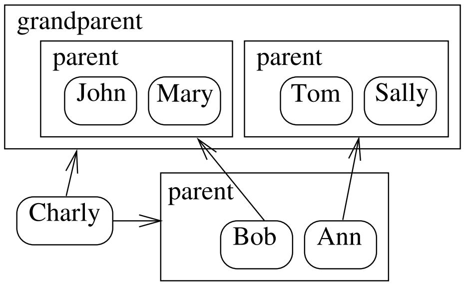
An advantage of our visual syntax is that it easily allows us to represent both alternatives by merging them in one single diagram:



Solving the query in this way corresponds to a breadth-first strategy (unlike Prolog's depth-first strategy). The unknown variable appears twice, showing that maybe two possible solutions can be computed. Textually this diagram would correspond to the following existential query:

$$\exists X \; X \in parent(Bob) \lor X \in parent(Ann) \; ?$$

Further transformation by matching the $parent$ boxes with their definitions in Figure 5 leads us finally to an *answer diagram*, where the original unknown variable appears fully instantiated. In this case it presents four different solutions to the original query, namely that either John, Mary, Tom or Sally is a grandparent of Charly:



To summarize, we have shown informally that it is possible to transform a query diagram into an answer diagram where no goal terms remain to be solved—i.e. a diagram that has no question marks— by means of two different basic inference steps. In both cases we match some box of the query diagram with its definition in the program. One case corresponds to predicates defined via the membership relation ($\in$). This inference step unifies the contents of the selected boxes instantiating —when necessary— the goal variables. The other case appears in predicates defined using the subset relation ($\subseteq$). This type of inference adds a new box to the query diagram capturing the transitivity of the subset relation with respect to set membership graphically.

Several advantages of this visual notation already arise in this simple example: We can keep track of the proof in our diagram during the query solving process in an elegant way, and different alternative answers can easily be presented within one single diagram.

3.2. Query Diagrams

Queries are not restricted to the simple (atomic) type of query diagram used in last Section, but are flexible enough to accommodate the different possibilities of existing logic programming languages, like the conjunction of several atomic queries. In Diagram (b) of Figure 4, for example, we can see such a more complex type of query diagram: we ask for grandparents of Charly who also are ancestors of Bob. Within it, the goal term is included in two boxes representing therefore conjunction (intersection) of properties. Diagram (c) of the same Figure shows another way to build complex queries. Here we are interested to know who has Ann as ancestor and Mary as grandparent. Finally, queries may also be posed using only fully instantiated terms, as it is the case of Diagram (d), where we ask wether Ann is or not a grandparent of Charly.

An *atomic query diagram* is formally defined as a diagram consisting of a predicate box, the arguments of this predicate box, and one element —either a variable, a constant or a compound term— which is graphically included in the predicate box. Of course, the structure of the term representing this element many be as complex as desired, and may also share common subterms with the arguments of the predicate. One way of interpreting this query diagram is to say that we ask for the membership of the element into the set corresponding to the set comprehension of the predicate. This atomic query diagram must have at least one term selected as goal, which will be indicated by a question mark. These goal term guides the inferences which lead to an answer diagram, when all question marks have disappeared due to unification.

General query diagrams can be built in two different ways: either having more than one element included in the predicate box, or by merging different atomic query diagrams into a new query diagram. This new complex query diagram will be the conjunction of the atomic ones. Note that in query diagrams only membership relations are allowed, inclusions of boxes into boxes are not taken into account when transforming the diagram. Furthermore, query diagrams differ from ordinary diagrams in that no box is marked using thick lines, since we are not defining a predicate.

3.3. Visual Inference Rules

In the previous subsection we have shown through an example the type of inferences that allow us to solve visual queries by means of diagram transformation. Let us now formalize these inferences as *visual rules of inference*. As we said before, we consider two types: inferences based on the membership relation and inferences based on the subset relation. Therefore, there are two visual inference rules corresponding to these two types of inference steps. In Figure 6 we find the visual rules which we call *membership rule* and *subset rule* respectively.

Technically, existential queries of the form

$$\exists X_1 \cdots X_n \; s_1 \in t_1 \wedge \cdots \wedge s_m \in t_m \; ?$$

are solved by refuting its negation

$$\forall X_1 \cdots X_n \; s_1 \notin t_1 \vee \cdots \vee s_m \notin t_m$$

in the context of resolution-based theorem proving. As already pointed out at the beginning of this section, since we are dealing with a transitive relation ($\subseteq$), we base the operational semantics of our visual language on the *chaining calculus* [3], whose inferences built in the axioms which define the special relation $\subseteq$.

As mentioned in Section 2.3, diagrams of a visual program actually are conditional membership or conditional subset statements, where conditions involve memberships only. Therefore the two visual inferences shown in Figure 6, the *membership rule* and the *subset rule*, actually correspond to *resolution* and *negative chaining* respectively,

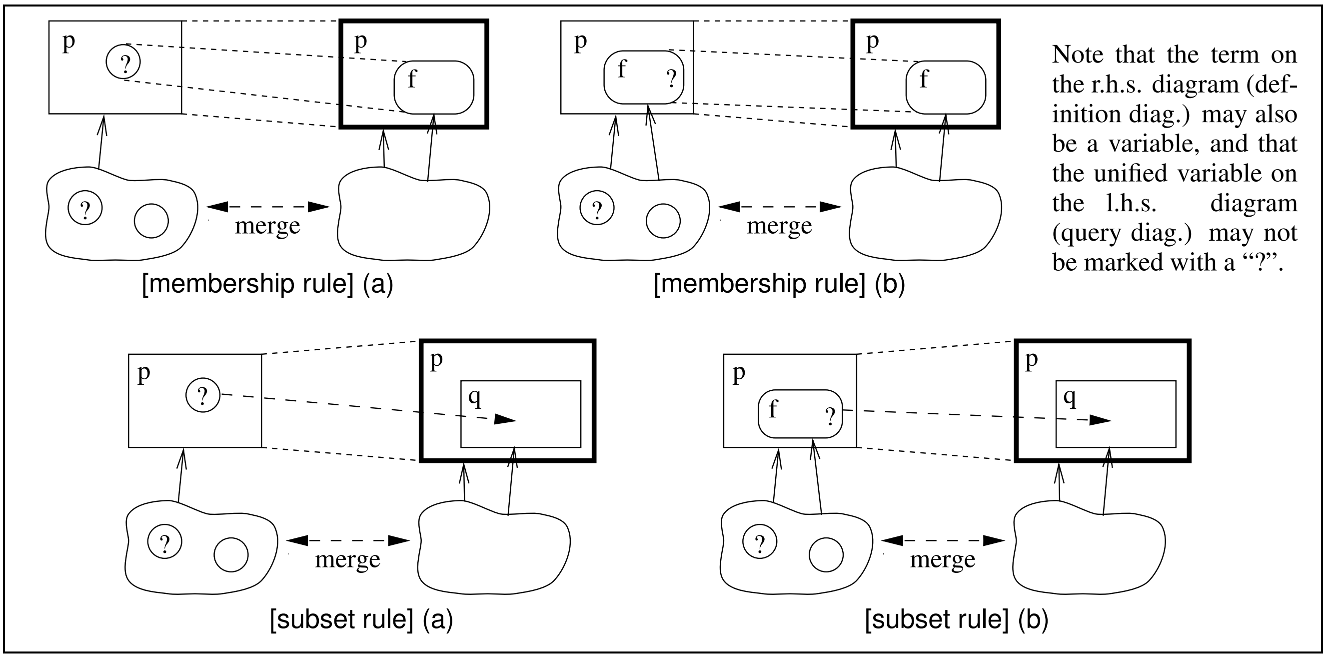


Figure 6. Visual Inference Rules

where conditional membership or subset statements are expressed as logically equivalent Horn clauses:

$$\text{(resolution)} \quad \frac{\underbrace{s \notin t \quad u \in v \vee C}}{C\sigma}$$
$$\sigma = mgu(s,u) = mgu(t,v)$$

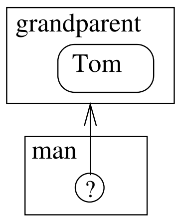
$$\text{(negative chaining)} \quad \frac{\underbrace{s \notin t \quad u \subseteq v \vee C}}{s\sigma \notin u\sigma \vee C\sigma}$$
$$\sigma = mgu(t,v)$$

However, although the rules are formally based on the (textual) chaining calculus, the visual inference mechanism is intrinsically different. Visual queries differ from textual ones, since they involve different kinds of query transformations, providing a new and interesting framework for exploring and developing visual inference systems. The main difference with regard to conventional operational semantics is that in our diagram we do not erase boxes (predicates) that have already been refuted, because we want to keep track of the trace of the proof. For instance, in Prolog every time we refute a goal it disappears from the query, finishing the inference process when the query is empty. For this reason in our queries we need to explicitly mark the goals with a question mark. Another difference is when we are applying diagrams which include more than one graphical item, and therefore various alternatives for solving a goal arise. As shown in Section 3.1, in this case we can represent all alternatives in one single query diagram by duplicating parts of it.

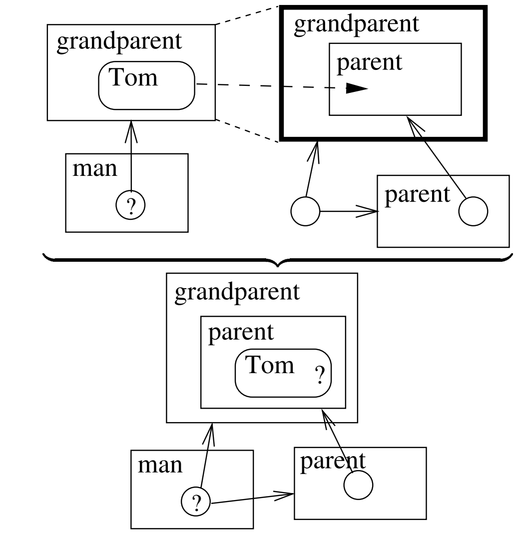
The two inference rules we present in this paper work in the same way: they match one of the boxes of the query with its definition in a visual program. One of the problems of this visual operational semantics is how to choose the box over which we are going to perform the next inference step, i.e. the order in which the rules are applied. Our approach to this problem is to apply first —whenever possible— the subset rule, and to choose boxes that are somehow related to one of the goals of the query, i.e. where one goal is included in the box, or it is in one of the arguments of the box, or even it is just a part of some term which is included in the box. When there is more than one box satisfying these criteria, we usually choose the one which has more fully instantiated arguments.

A very important part of applying the visual inference rules of Figure 6 is to merge in a "convenient" way the subgraphs of the boxes being matched. We need to match one by one the arguments of these boxes, instantiating variables when necessary, and we also need to match other common boxes that appear in both diagrams. A more elaborate treatment of the merging (or unification) algorithm is beyond the scope of this paper, but we show how these visual inference rules are applied by means of an example:

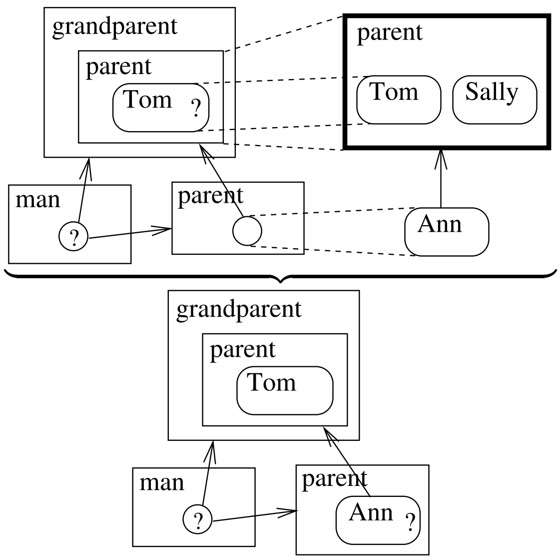
Suppose we want to know if Tom has a male grandson (using the visual program of Figure 5). This would be represented by the following query diagram:



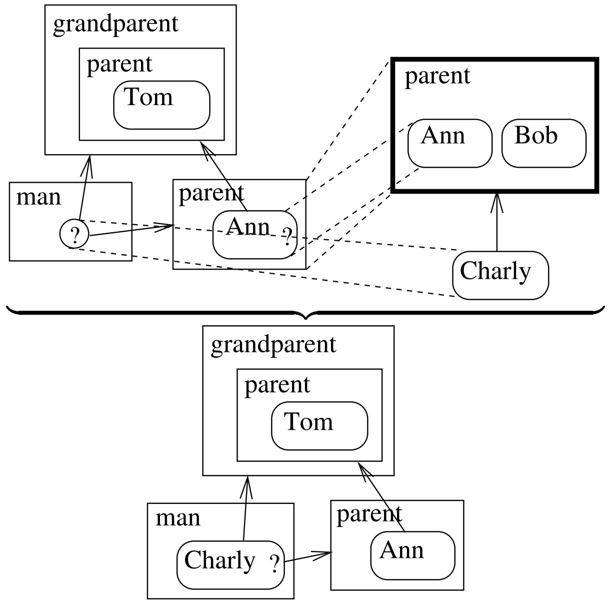
At this point we have two different possibilities: to apply the subset rule to the *grandparent* box or to apply the membership rule to the *man* box. We choose the first option:



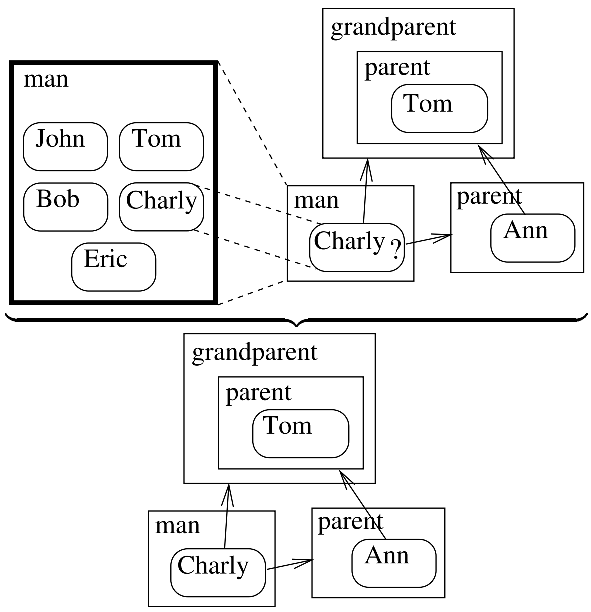
But in the resulting query diagram there appears a new goal term membership, that of Tom into one of the *parent* boxes. This inclusion has to be checked before we can qualify the resulting diagram as a valid inference, otherwise we will have to backtrack. We match the parent box with its definition in Figure 5.



This inference step shows us that the previous goal, Tom, is momentarily valid, but it induces a new goal by instantiating a variable which is included in the other *parent* box. Therefore we have to check this membership of Ann into *parent*:



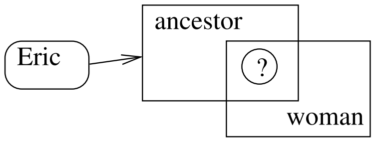
Again the previous inference step using the membership rule induces a new goal because it instantiates the goal variable which is included into the *man* box. We finally obtain the answer diagram by applying the membership rule to the man box, in order to check that Charly is really a man:



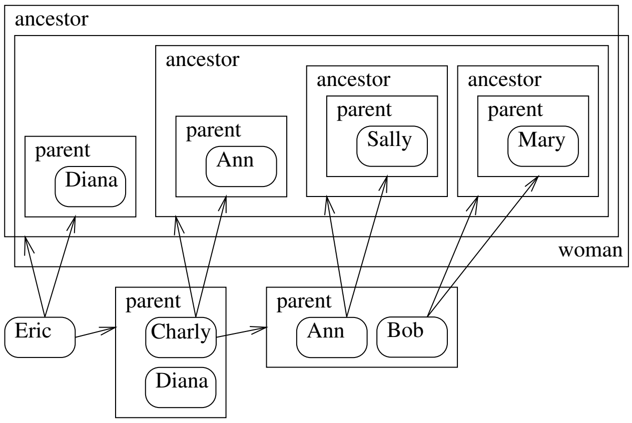
Therefore this is a correct answer diagram, and Charly is a male grandson of Tom.

3.4. Diagram Simplification

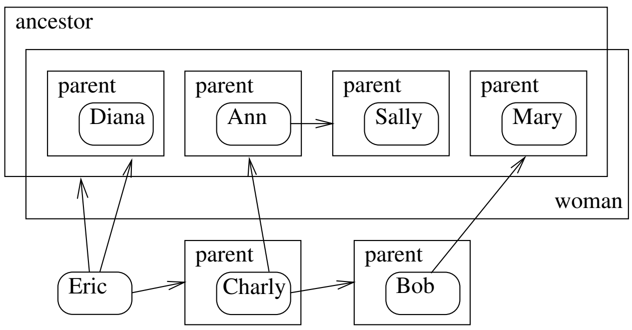
Not always this methodology of diagram transformation produces clear and easy-to-read diagrams. This is the case when recursive definitions are involved. Consider for example the definition of the ancestor relation given in Figure 5. Solving the query:



will give us the female ancestors of Eric. The problem is that the *ancestor* relation is recursive and leads us to the following difficult to read answer diagram:



This diagram has already been simplified since it only has deduction branches which contain some valid result, i.e. failures that have provoked backtracking are already deleted from the diagram. However we think it is still too difficult to read. We propose three diagram simplification rules: (1) The recursive definition appearing several times in the answer diagram can be reduced to only one, the original one in the query. (2) We can discard repeated structures appearing more than once in the same diagram. (3) Finally, we can also eliminate elements that do not produce any valid solution, like Bob and one of the occurrences of $Diana$ in the former answer diagram. Therefore, a better representation of the answer to the query, whilst keeping the relevant information about the trace of the proof would be:



4. Conclusions and Work in Progress

We have shown that it is possible to have a fully visual operational semantics for a visual declarative programming language. We have also demonstrated that our visual syntax is useful to represent queries and answers, keeping track of the trace of the proof and being able to show different alternatives in a single diagram. However, this is only the first step towards a fully visual operational semantics.

Immediate work is going to focus on proving the completeness of the inference system presented in this paper — soundness is obvious—, and enhancing it in order to cope with other features of the VL'96 visual language [12] like predicate composition. We would also like to explore the use of other programming constructions, like for instance partitioning constraints [13] or richer element representations, to improve the expressive power of the language. Other work we contemplate is to perform larger experimentation and to develop a specific programming methodology for this visual logic programming language.

Finally, we are also interested in studying how the use of a visual syntax changes the formal properties of the operational semantics. We want to know if there are other advantages —apart from the obvious visual ones— of using visual syntax in declarative programming.

References

[1] J. Agustí, J. Puigsegur, D. Robertson, and W. M. Schorlemmer. Visual Logic Programming through Set Inclusion and Chaining. In *CADE-13 Visual Reasoning workshop*, 1996.

[2] G. Alwein and J. Barwise, editors. *Logical Reasoning with Diagrams*. Oxford University Press, New York, 1996.

[3] L. Bachmair and H. Ganzinger. Rewrite Techniques for Transitive Relations. In *Proc., 9th IEEE Symposium on Logic in Computer Science*, pages 384–393, 1994.

[4] W. Citrin, R. Hall, and B. Zorn. Programming with Visual Expressions. In *Proc. of the 11th IEEE Symposium on Visual Languages*, Darmstadt, Germany, 1995.

[5] D. Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[6] D. Ladret and M. Rueher. VLP: a Visual Logic Programming Language. *Journal of Visual Languages and Computing*, 2:163–188, 1991.

[7] J. Levy and J. Agustí. Bi-Rewriting, a Term Rewriting Technique for Monotonic Order Relations. In C. Kirchner, editor, *Rewriting Techniques and Applications*, LNCS 690, pages 17–31. Springer-Verlag, 1993.

[8] J. Levy and J. Agustí. Bi-rewrite Systems. *Journal of Symbolic Computation*, 22:1–36, 1996.

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

[10] M. A. Najork. Programming in Three Dimensions. *Journal of Visual Languages and Computing*, 7:219–242, 1996.

[11] L. Pau and H. Olason. Visual Logic Programming. *Journal of Visual Languages and Computing*, 2:3–15, 1991.

[12] J. Puigsegur, J. Agustí, and D. Robertson. A Visual Logic Programming Language. In *Proc. of the 12th IEEE Symposium on Visual Languages*, Boulder, Colorado, 1996.

[13] L. Spratt and A. Ambler. A Visual Logic Programming language based on Sets and Partitioning constraints. In *Proc. of the 9th IEEE Symposium on Visual Languages*, 1993.