User-Defined Visual Languages for Querying Data

Isabel F. Cruz

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-93-58
December 1993

User-Defined Visual Languages for Querying Data

Isabel F. Cruz*
Department of Computer Science
Brown University
Providence, RI 02912-1910
ifc@cs.brown.edu

Abstract

DOODLE is the first language to be proposed for querying object-oriented databases with user-defined visual languages. A DOODLE program defines a visual language in a by example fashion. In this paper we continue the preliminary exposé on DOODLE (SIGMOD'92). We refine the language, and provide complex examples that illustrate the extensive capabilities of DOODLE in a variety of applications. The contributions of DOODLE to other areas of Computer Science, such as constraint languages and graph drawing are also outlined. DOODLE explores the limits of declarative languages by making data and its manipulation visual, while staying within the bounds of the formal database field. The formal basis of DOODLE motivates the study of visual expressiveness and points to further research.

1 Introduction

It has a been a long-standing claim that pictures convey information in ways that text and tables cannot, as illustrated by the books of Tufte [Tuf83, Tuf90] and Bertin [Ber83]. In this paper a *picture* is a set of graphical symbols, and a *visual language* is a set of pictures. All the pictures in a visual language obey the same conventions. In our approach visual languages are user-defined by means of a DOODLE (for *Draw an Object-Oriented Database LanguagE*) program. The primary aim of DOODLE is to formally define visual languages for representing facts in an object-oriented database. Also, with DOODLE the user can query the database using the same conventions as for displaying the facts.

DOODLE differs from other visualization proposals (e.g., [Bor81, Wyk82, Rei90, GC90, Kam89]) in that we look at data and its visual manipulation from a database point of view, and stay within the bounds of the formal database field. DOODLE also differs from other query languages for object-oriented databases (e.g., [GSKZ85, Cru90, GPV90]) in that the user defines how to visualize and query the data. A related approach is PROTEUS [AEM86]. However, there are important differences: In DOODLE, visualizations (mappings from data to pictures) are specified visually, and the query languages are user-defined, while in PROTEUS visualizations are specified textually. PROTEUS is object-oriented in the usual sense, while in DOODLE the object-oriented concepts are extend to visual languages.

A preliminary exposé of DOODLE appeared in [Cru92]. Since then we have considerably extended DOODLE. For instance, we have added a form of multiple inheritance through visual language composition. As well, we have designed an important component of DOODLE: *the U-term language*. This is the part of DOODLE with which the user specifies visually the display of data. An important characteristic of this language is the small set of primitives, and their genericity. The U-term language also fits well with the object-oriented characteristics of DOODLE, and it supports the specification of a variety of displays (e.g., graphs, bar charts, pie charts, plot charts).

The ability to specify a variety of displays and the new capabilities of DOODLE make complex examples possible, which further demonstrate the usefulness of DOODLE. We include examples that perform *visual aggregation*, *visual grouping* and *visual recursion*. Meta-queries can also be expressed in a visual way.

*Partial support was given by ARPA order 8225, ONR grant N00014-91-J-4052.

Another important application of DOODLE resides in its capability to express graph layout. (e.g., that a tree is to be displayed in an downward fashion). DOODLE is the only database visualization language with such a capability. The visual specification of graph layout is one of the contributions of DOODLE to other areas of computer science, in this case to graph drawing [DETT93]. The U-term language is also a contribution to the constraint programming area. In using the technology of declarative and object-oriented query languages and integrating it with other areas, we explore the limits of declarative query languages [AV87, Abi88, AK89, KLW90, GZG92, BK93].

The semantics of DOODLE is formally defined and given by F-logic [KLW90]. The formal basis of DOODLE, and the extension of the concept of database querying provides a foundation for a formal study of visual languages.

We organize the rest of the paper as follows. In Section 2 we give an overall up-to-date perspective of DOODLE. Section 3 provides an overview of the U-term language, with a detailed example in the domain of graph drawing. In Section 4 we give the semantics of the extensible capabilities of DOODLE. Complex examples of the language are presented in Section 5. Section 6 discusses some of the expressiveness issues that arise with the querying of data by its visual representation. The architecture of a system that implements DOODLE is proposed in Section 7. In the final section we present conclusions and directions for future research.

2 DOODLE

2.1 DOODLE Terms and Objects

There are several kinds of terms and objects that DOODLE deals with. We give informal definitions below. Figure 1 depicts some of these objects.

F-terms. These are F-logic terms [KLW90], such as

$$\mathrm{X} : \mathit{procedure}[\mathrm{name} \rightarrow \text{"}\mathsf{draw}\text{"}; \mathrm{callsFrom@myModule} \rightarrow \{\mathrm{Y} : \mathit{procedure}\}].$$

In this term, X and Y are variables that represent object identities, and *procedure* is a class (e.g., X belongs to class *procedure*). name and callsFrom are methods. The former method maps the object identity X to the a string that represents the name of the procedure. The latter method has an argument (moduleId), and maps the pair (X, *myModule*) to a set of objects of class procedure (the set of procedures that are called by X and are in *myModule*).

In the F-logic model, classes are organized in hierarchies and there is method inheritance and overriding.

V-terms. These are visual representations of the F-logic terms. In Figure 1(i) we represent the V-term for the F-logic term above).

Database Objects. These are objects in the object-oriented extensional database (EDB). They are ground F-terms. Objects belong to a class, and have methods.

U-terms. These are pictures in the U-term language that are syntactically correct. U-terms specify by example the display of the database objects. U-terms are formed of the following kinds of graphical symbols: *prototypical symbols*, *macro symbols*, *key symbols*, and *generic symbols*. In Figure 1(ii) there are three key symbols: the dashed boxes and the dotted box (see Figure 2), and the macro symbol *contains* (expressed by the four arrows).

Visual Objects. Visual objects are instances of the visual classes, and correspond to the graphical objects that can be drawn in the screen. We consider the following visual classes: *box*, *diamond*, *circle*, *text*, *circle*, *sector*, *straightLine*, *arrow*, and *doubleArrow*. Such a limited set is sufficient for the kinds of displays of the database that we consider (e.g., graphs, bar charts, pie charts). Sets of visual objects that are considered in related work are comparable to the one we consider (e.g., [Gol91, Wyk82]), or more restricted (e.g., [Mac86]).

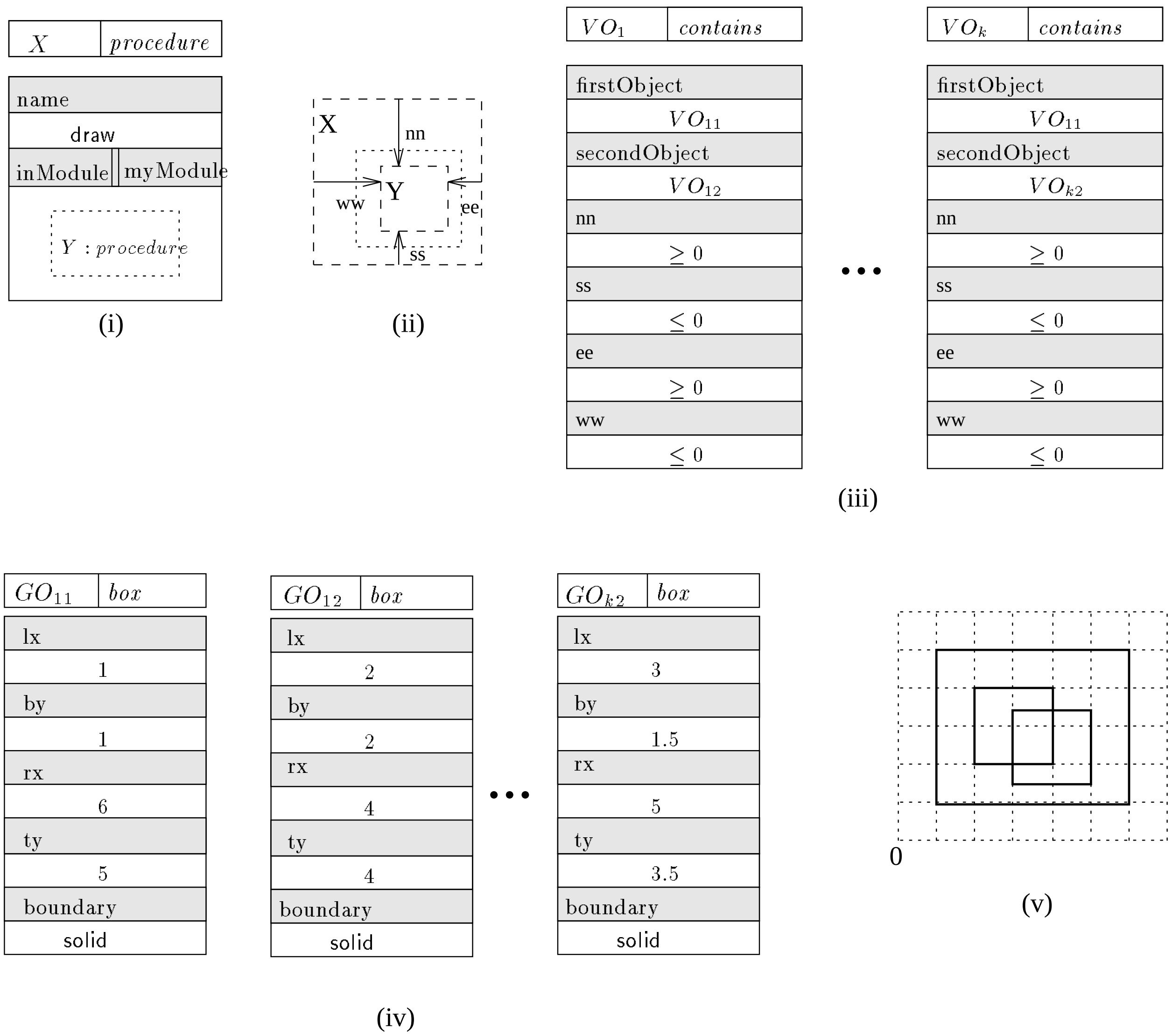


Figure 1: DOODLE Objects and Terms.

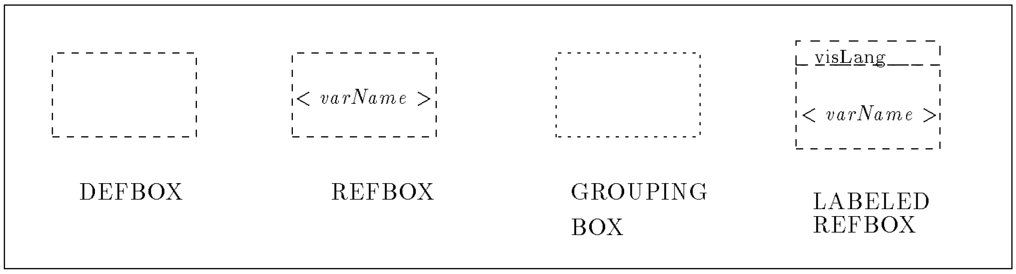


Figure 2: Some key symbols in DOODLE.

Visual Constraint Objects. These are objects of two kinds: *length constraint objects* and *overlap constraint objects*. *Length constraint objects* express lengths in one object (e.g. the object's width), or between two objects (e.g., the distance between two points, one in each object). Given two visual objects, an *overlap constraint object* specifies which object is on top of the other object. Visual constraint objects have an F-term (or V-term representation). In Figure 1(iii) we give the V-terms that correspond to the object of class *contains* the U-term. Notice that for each X there are as many visual objects as there are objects Y.

Graphical Objects. These are objects that are represented in the screen (Figure 1(v)). Their representation using V-terms is as exemplified in Figure 1(iv).

2.2 Visual Languages and DOODLE

A *visual language* is a set of pictures that are specified by a DOODLE program. A DOODLE visual program is a set of visual rules. The order of the rules is immaterial. Visual rules are vertically divided. The left-hand side contains F-,V-, or U-terms that are being defined, and the right-hand side contains "known-terms". We read each visual rule in the following way: if the right-hand side is true, then the left-hand side is also true.

The program of Figure 3, is an example of a DOODLE program that defines a higraph [Har88]. Figure 1. In Figure 3 there are two kinds of terms: F-terms and U-terms. All the terms to the right of the double

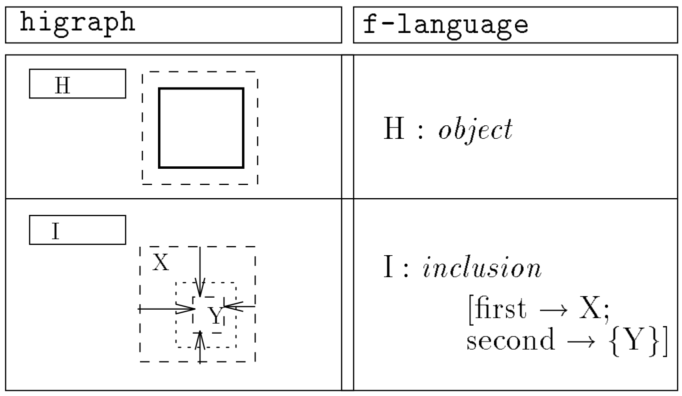


Figure 3: DOODLE program that defines **`higraph`**.

bar are F-terms, hence the box $\boxed{\texttt{f-language}}$. On the left-hand side, the U-terms are used to define the visual language **`higraph`**, as indicated by $\boxed{\texttt{higraph}}$. Informally, this program and the query of Figure 4 specify that for the database facts that make the F-terms true, graphical objects similar to the U-terms on

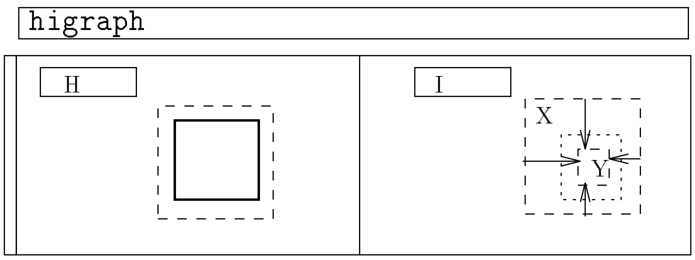


Figure 4: DOODLE query.

the left-hand side of Figure 3 will be drawn. The first rule states that any object M in class *module* is to be displayed by a box. The second rule states that any object I in class *inclusion* is to be displayed by a containment between the object that graphically represents the database object X and each graphical object that represents an element Y in the set. Refboxes allow for U-terms to refer to U-terms that were defined elsewhere. For example, in the objects of class *inclusion*, the refboxes make reference to the U-term that specifies the display of objects of class *object*. This reference is made possible by the use of a defbox in the rule that defines the visualizations of objects of class *object*. The defbox indicates the U-term (or part of the U-term) that can be referred to by another rule. The roles of defbox and refbox are analogous to the concepts of procedure definition and of procedure call. The defbox specifies the display, and the refbox "calls" it.

Labeled refboxes are a generalization of refboxes. While (simple) refboxes make a call to the current visual language (**higraph** in the example), labeled refboxes make a call to another visualization or to an expression relating visualizations.

2.3 Semantics of DOODLE

The semantics of DOODLE programs is given by the semantics of the equivalent F-logic program. The F-logic program is achieved by mapping each visual rule to one F-logic rule. In this translation, each U-term is mapped to a set of F-logic objects.

The program of Figure 3 defines a mapping from database objects to visual objects. We call such mapping a *visualization*. A set of F-logic objects that represent the visual objects is a minimal model of the equivalent F-logic program:

$$
\begin{aligned}
&\mathrm{H}[\mathrm{display}@\mathtt{higraph} \rightarrow \{\mathrm{vis(H)} : \mathit{box}\}; \mathrm{defbox}@\mathtt{higraph} \twoheadrightarrow \{\mathrm{vis(H)}\}] \longleftarrow \mathrm{H} : \mathit{object}.\\
&\mathrm{I}[\mathrm{display}@\mathtt{higraph} \rightarrow \mathrm{vis(I)} : \mathit{contains}[\mathrm{out} \rightarrow \mathrm{U}; \mathrm{in} \rightarrow \mathrm{agg(I)} : \mathit{visAgg}[\mathrm{visMembers} \rightarrow \{\mathrm{V}\}]]] \longleftarrow\\
&\qquad\qquad \mathrm{I} : \mathit{inclusion}[\mathrm{first} \rightarrow \mathrm{X}; \mathrm{second} \rightarrow \{\mathrm{Y}\}],\\
&\qquad\qquad \mathrm{X}[\mathrm{defbox}@\mathtt{higraph} \rightarrow \mathrm{U} : \mathit{visualObject}],\\
&\qquad\qquad \mathrm{Y}[\mathrm{defbox}@\mathtt{higraph} \rightarrow \mathrm{V} : \mathit{visualObject}].
\end{aligned}
$$

Each visual object and visual constraint object has an identity that is determined by the object or objects it gets mapped from. For example, in this case for each object H there is a a visual object of class *box*, as denoted by the object constructor vis(H).

A DOODLE program describes one picture, if the picture is completely specified (that is the exact placement of the graphical objects, dimensions of the objects, distances between objects, etc., are given). In general, though, a DOODLE program describes a set of pictures. Any two pictures in the set differ in one or more of the picture attributes that were not completely specified by the DOODLE program. In Figure 1 for each visual object (e.g., VO_{11}) there can be a set of graphical objects that form an equivalence class (e.g., GO_{11} is one graphical object in the equivalence class that is specified by VO_{11}).

3 U-term language

There are attributes that are common to all visual objects, and therefore belong to *visualObject* (the superclass of all visual classes), e.g., *thickness*, *color*, and *texture*. Attributes can have default values in the class *visualObject* that can be overridden in the subclasses.

There are two other set-valued attributes: *landmarks* and *anchorpoints*. *Landmarks* are used to give dimensions to the objects, and to place objects relatively to other objects. Landmarks can belong to the following classes: *landmark*, *cartesianLandmark*, *polarLandmark*, and *lineLandmark*. *Anchor points* are user-defined landmarks. From now on we use *point* to denote a landmark or an anchor point. Points can be described in two formats: *cartesian* or *polar* (see Figure 5).

Visual constraint objects are instances of *visual constraint classes*. The constraints are expressed in terms of the objects' points. We show an application of the U-term language to the visualization of binary trees. The U-term of Figure 6 is used to specify the display of a binary tree in the following way:

- Downward, that is, a child is placed below its parent, and the edges point down.
- The X-coordinate of each node is proportional to the inorder rank of that node.
- The Y-coordinate of each node is proportional to the distance of that node from the root. This layout ensures that no nodes overlap or edges cross.

The following points explain some of the features of Figure 6:

- The U-term contains the following prototypical symbols: two arrows, an invisible refbox (in light grey), and a circle (in darker grey).

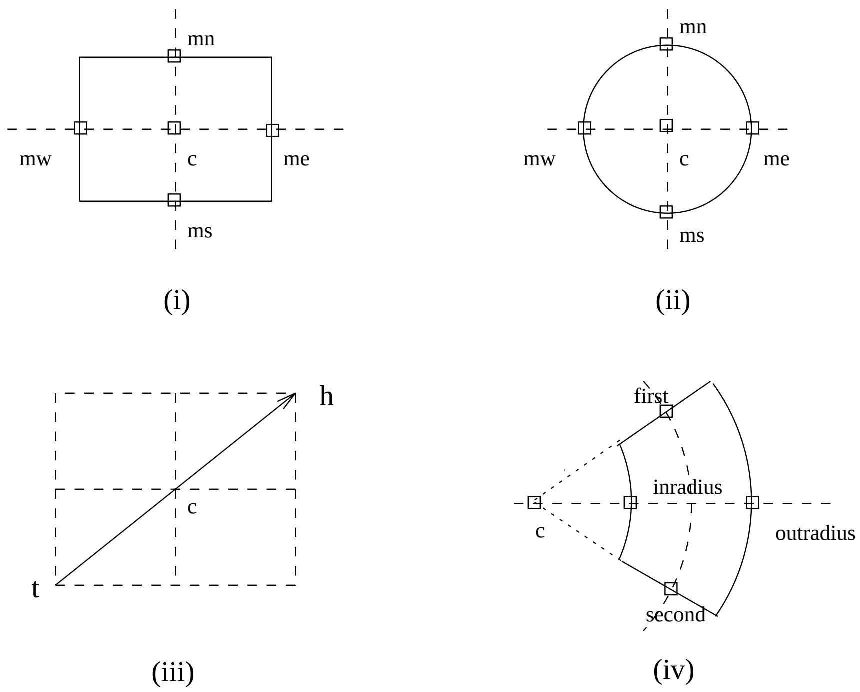


Figure 5: Cartesian landmarks: (i) on a box; (ii) on a circle; (iii) on a line. Polar landmarks: (iv) on a sector.

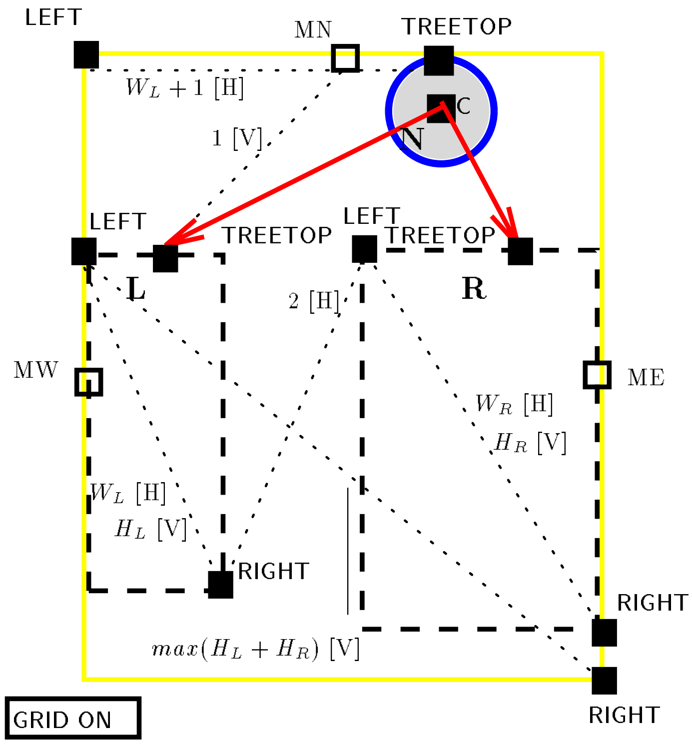


Figure 6: U-term that specifies the display of a binary tree in an downward fashion.

- The invisible box expresses the concept of bounding box (that is the tightest box) around the whole tree. This bounding box contains most of the landmarks and anchorpoints that are needed to define these binary trees.

- An overlap constraint is also used (denoted by the dark grey color of the circle), which specifies that the circle is the top object. In this way, the edges are specified to come out of the center of the node, but are hidden in the part that coincides with the node.
- Length constraints (represented as dotted lines between points) specify that each parent node is horizontally placed in between its subtrees. Other configurations are possible by changing the length of the constraint that positions the root (e.g., centering the root between its children). Other zero-length constraints are visually represented by the macro symbol GRID ON. This symbol specifies that two points on the same horizontal (vertical) line on a grid have the same vertical (horizontal) coordinates.
- The width of the tree's bounding box is given by the sum of the widths of the bounding boxes of the subtrees plus one.
- The height of the bounding box of the tree is equal to one plus the maximum height of the bounding boxes of the two subtrees.

4 Using Visual Languages to define Other Visual Languages

Visual languages are a central concept of DOODLE. In this section we present the different kinds of manipulation of visual languages: *inheritance, transformations* and *composition*. In this way we extend the object-oriented concepts that usually apply to classes and objects to the visual languages themselves. The following is a summary of these DOODLE capabilities, which have been considerably extended since [Cru92].

4.1 Inheritance of Visual Languages

The user can define that a visual language inherits from another visual language, with the possibility of redefining the visualization for objects of one or more classes (*overriding*), and the possibility of adding visualizations for more classes in the new visual language. To specify that `V` inherits from a single visual language `V'`, the user writes `V:V'`. We keep this information in F-terms of the form:

$$\mathtt{V}[\text{super} \rightarrow \mathtt{V}']$$

The hierarchy of visual languages is a tree. Suppose we want to visualize objects of class c with visualization `v`. The simplest possibility is that `V` has been defined for c. Otherwise, if the visualization for `V` for c has not been explicitly specified, the following possibilities can occur:

- If the visualization `V` for C' is known (where C' is the closest superclass of C for which the visual language `V` has been defined), objects of class C will be visualized as objects of class C' (see Figure 7(i)).
- If the visualization `V` for C (or any superclass of C) is not known, objects of class C are visualized using `V'`, where `V:V'`(see Figure 7(ii)).
- If the visualizations `V` and `V'` are not known for C or any of its subclasses, a run-time error occurs (see Figure 7(iii)).

The semantics of visual inheritance is embodied by the following meta-rule, as expressed in F-logic:[1]

$$\begin{aligned} \mathrm{X} : C[\text{display@}\mathtt{V} \rightarrow \mathrm{Y}] \longleftarrow & \neg \mathrm{X} : C[\text{display@}\mathtt{V} \rightarrow \mathrm{Y}], \\ & \mathtt{V}[\text{super} \rightarrow \mathtt{V}'], \\ & \mathrm{X} : C[\text{display@}\mathtt{V}' \rightarrow \mathrm{Y}]. \end{aligned}$$

Notice that

- The term $\neg \mathrm{X} : C[\text{display@}\mathtt{V} \rightarrow \mathrm{Y}]$ is only true if visualization `V` is not defined for C or any of its superclasses.

[1] For simplicity we omit the method defbox.

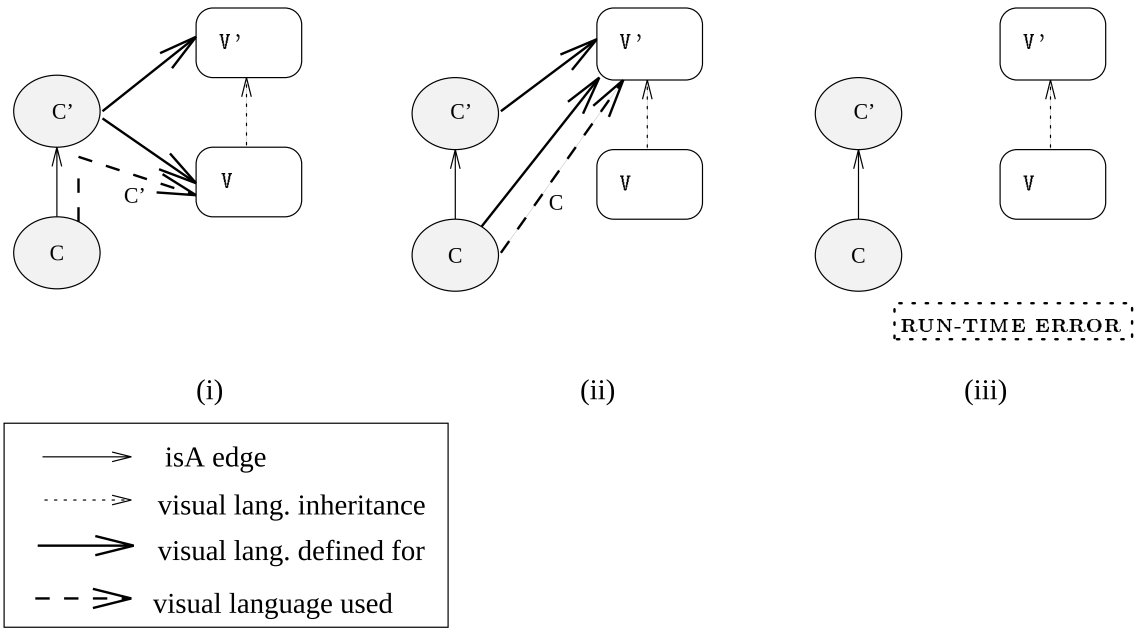


Figure 7: Inheritance of Visual Languages.

- Other display methods that may have been defined for class C, do not override the display method that is inherited from a superclass of C for the visual language `V`. That is, if there is a method display@`T` (which is defined for C), this method does not override the method display@`V`, because of the pointwise inheritance of methods in F-logic.

  The above strategy can be easily generalized to consider more than two visual languages, such as, `V:V':V''....`

4.2 Transformation of Visual Languages

A transformation is a mapping from pictures in a visual language to pictures in another visual language. Typically, as in the case of visualizations, we specify for each database class, how visual objects in a visual language `V` map into visual objects in another visual language `V'`. In order to define a new visual language, F-terms, or U-terms in other visual languages than `V` may be used. As well, visual objects in visual language `V'` can be defined in terms of the visual language `V'` itself.

4.3 Multiple Inheritance and Disambiguation

In DOODLE we prevent a visual language from inheriting from more than one visual language. However, in our data model, multiple inheritance of classes is possible. This may lead to ambiguity in the visualization for objects of a class C, if this class has more than one superclass for which the visual language `V` that we want to use is defined. For example:

X : C'[display@`V` $\rightarrow$ Y : $visual'$]
X : C''[display@`V` $\rightarrow$ Y : $visual''$]

and

$C : C'$
$C : C''$

The user can specify from which of the classes it inherits `V`, with the following syntax:

inherit `V` **from** *C'*.

To implement this there are two possibilities that are offered by F-logic [KLW90]:

$$\mathrm{X} : Y[\mathrm{param}(\mathrm{display},\ \mathrm{Y})@\mathtt{Z} \rightarrow \mathrm{W} : \mathit{visualObject}] \longleftarrow \mathrm{X} : Y[\mathrm{display}@\mathtt{Z} \rightarrow \mathrm{W} : \mathit{visualObject}]$$

In this case we are parameterizing the display method by the class to which it belongs. The method 'param' can now be used instead of the method display, with class *C'* as an argument to the method param.

The other possibility is expressed by

$$\mathrm{X} : C[\mathrm{display}@\mathtt{Z} \rightarrow \mathrm{W} : \mathit{visualObject}] \longleftarrow \mathrm{X} : C'[\mathrm{display}@\mathtt{Z} \rightarrow \mathrm{W} : \mathit{visualObject}]$$

We make the method display of class C behave like the method display of superclass C'.

4.4 Composition of Visual Languages and Pointwise Multiple Inheritance

Point-wise multiple inheritance is achieved with a labelled refbox. The label of the refbox is a composition of two or more visual languages. Such a composition is expressed by an expression on visual language names. For example, the expression $\mathtt{v} \otimes \mathtt{v}'$, where the composition operator $\otimes$ is defined by the user. We present next the semantics for any composition operator *op* (when we compose two visual languages):

$$\begin{aligned}
&[\mathrm{display}@\mathrm{op}(\mathtt{V},\ \mathtt{V}'),\ \mathrm{X} \rightarrow \mathrm{vis}(\mathrm{X}) : \mathit{visualObject}_3[\mathrm{attr}_1 \rightarrow \mathrm{o}_1,\ \ldots,\ \mathrm{attr}_n \rightarrow \mathrm{o}_n]]\\
&\qquad \longleftarrow \mathrm{X} : \mathit{object}[\mathrm{display}@\mathtt{V},\ \mathrm{X} \rightarrow \mathrm{Y} : \mathit{visualObject}_1[\mathrm{attr}_1 \rightarrow \mathrm{Z}_1,\ \ldots,\ \mathrm{attr}_n \rightarrow \mathrm{Z}_n]],\\
&\qquad\quad \mathrm{X} : \mathit{object}[\mathrm{display}@\mathtt{V}',\ \mathrm{X} \rightarrow \mathrm{U} : \mathit{visualObject}_2[\mathrm{attr}_1 \rightarrow \mathrm{W}_1,\ \ldots,\ \mathrm{attr}_n \rightarrow \mathrm{W}_n]].
\end{aligned}$$

where $\mathit{visualObject}_3$ is a visual class that inherits from $\mathit{visualObject}_1$ and from $\mathit{visualObject}_2$. We note that for each n we need a different rule. The oids $\mathrm{o}_i, 1 \leq i \leq n$ are defined in the following way:

$$\mathrm{o}_i : \mathit{composition}[\mathrm{operation} \rightarrow \mathrm{op}; \mathrm{args} \rightarrow \{\mathrm{Z}_1,\ \mathrm{W}_1\}; \mathrm{res} \rightarrow \mathrm{R}_1]$$

5 Examples

5.1 Basic Displays

It is convenient to have some basic displays (user-defined) for entities that appear frequently in the database (e.g., strings) or for graphical objects that are displayed frequently (e.g., nodes in a graph). For example, the visual language for `labeledNode`, which is defined in Figure 8. This visual language uses another user-defined

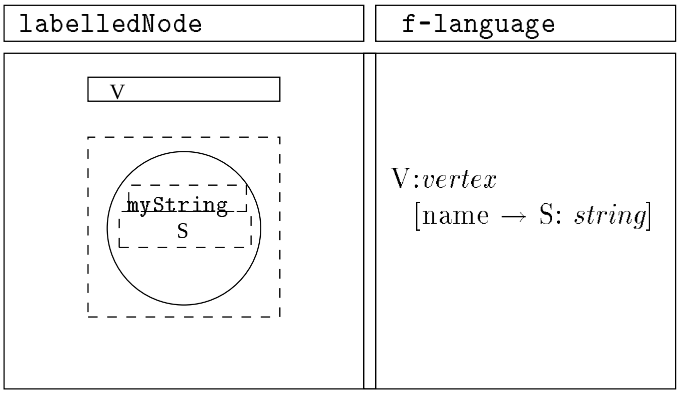


Figure 8: Visual program that defines `node.`

visual language for the display of strings (`myString`).

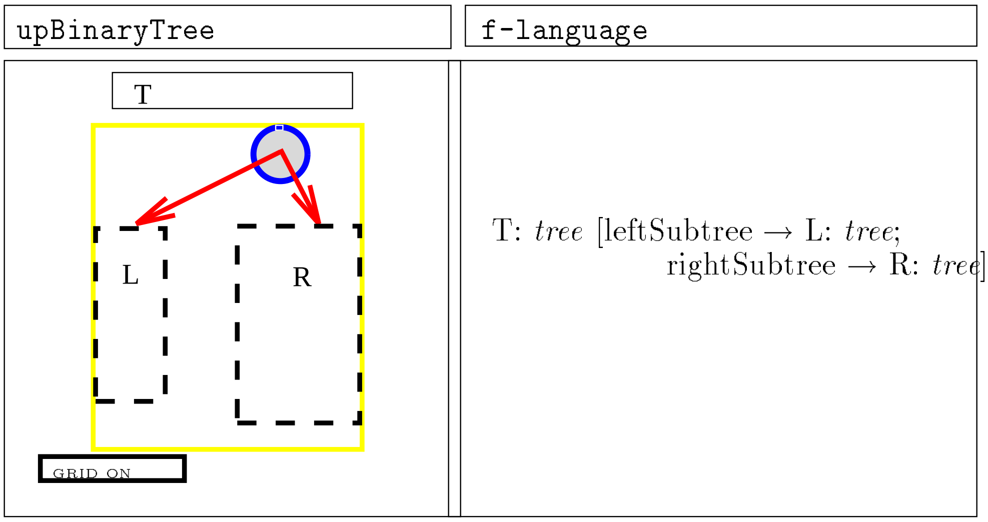


Figure 9: Visual program that defines `upBinaryTree`.

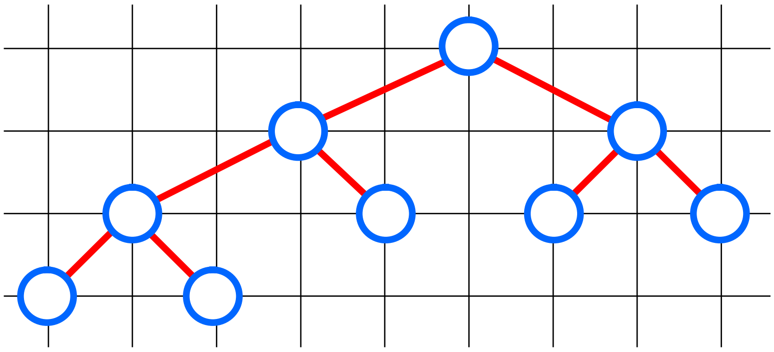


Figure 10: Downward layout of a binary tree specified by the program of Figure 9.

5.2 Tree Layout

The program of Figure 9 shows how we can define the downward drawing of a binary tree, which we represent in Figure 6. The U-term on the left-hand side is the same of Figure 6. The DOODLE program that specifies the downward layout of the binary tree has the important property that isomorphic subtrees are displayed by the same layout (up to translation).

With small modification we can specify the drawing of the tree in an inclusion form. The DOODLE program is in Figure 11, and the layout is in Figure 12.

5.3 Graph Transformation

This example illustrates a transformation between graphs (see Figure 13). The idea is that we take a graph which is displayed with a visual language `graph`, and whose transitive closure is displayed with visual language `tcGraph`, and find the nodes that are in the same strongly connected component as one of the nodes (the node that is labeled $'1'$ in the first visual rule). Nodes in the same strongly connected component are placed inside a "supernode" (second visual rule). We want to maintain the connections between the nodes inside the strongly connected component and from (or to) these nodes to the outside nodes (see third and fourth rules). Notice that the symbol '•' specifies that there will be a single object of class *sameSCC*, whose object identity is functionally dependent on the identity of the node labeled $'1'$.

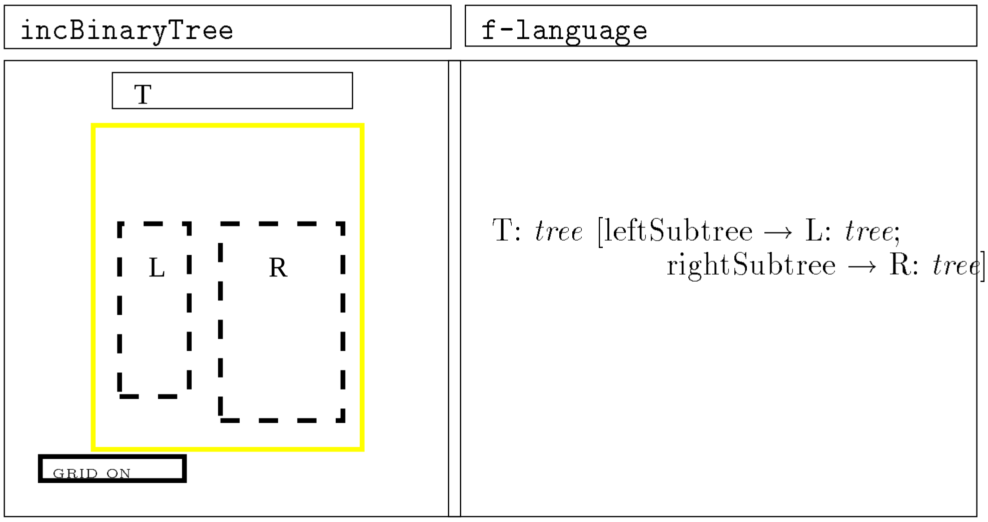


Figure 11: Visual program that defines `incBinaryTree`.

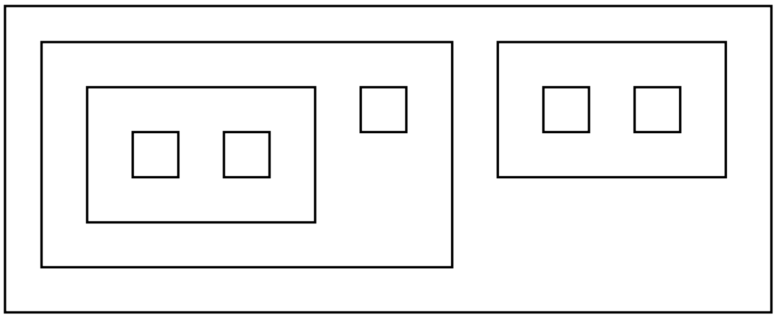


Figure 12: Inclusion layout of a binary tree specified by the program of Figure 11.

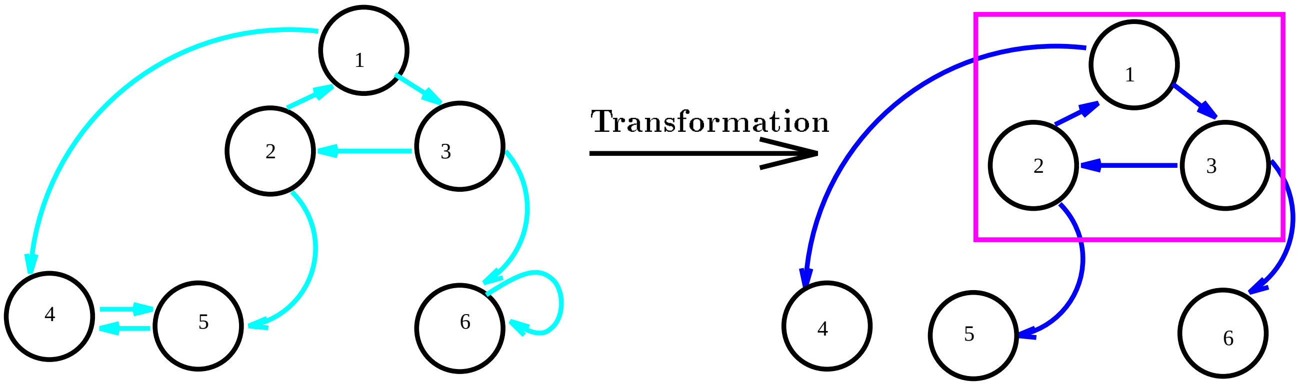


Figure 13: Transformation from a graph to another graph, emphasizing one strongly connected component.

5.4 Visual Recursion.

In this example, we show a visual language can be defined in terms of itself. The program of Figure 15 specifies that objects of class *root* are represented by a box. If the parent object is represented by a box, then the child object is represented by a diamond that is inscribed in the parent box. If the parent node is represented by a diamond, then the child node is represented by a box that is inscribed in the parent diamond. The inscription is specified with the help of transparent boxes (in light grey). These boxes

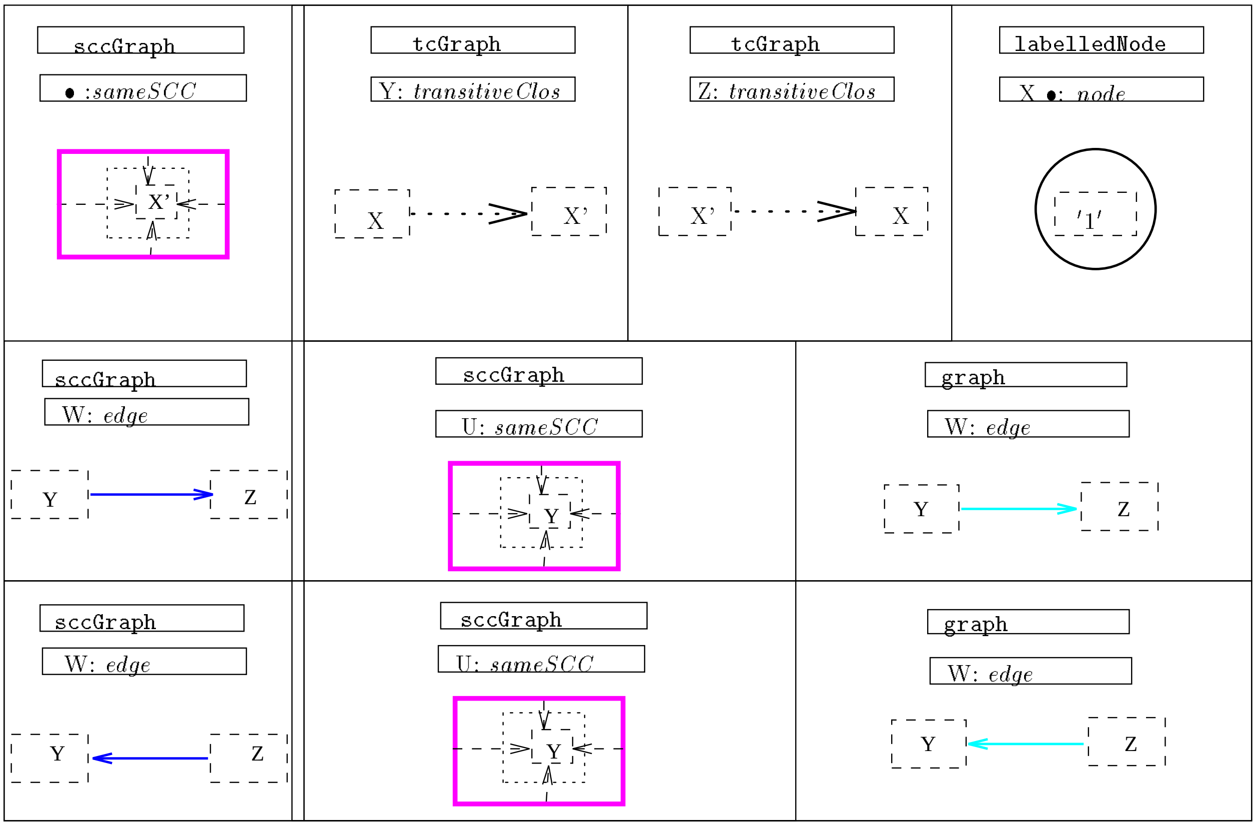


Figure 14: Transformation from a graph to another graph, emphasizing one strongly connected component.

have landmarks that make the specification of the position the graphical objects possible. In Figure 16 we represent a database and a picture that results from applying the DOODLE program of Figure 15 to that database.

5.5 Visual Summarization

In the visual program of Figure 17, we illustrate both recursion and the capacity of summarizing data. We are building a new bar chart by adding up the heights of the bars of the same color. More specifically, we are adding the height of the first element of the list, which has color C, to the summarized height of the elements in the rest of the list that also have color C. The x-coordinate is determined by the color of the bar (this coordinate is made 'proportional' to the color, and expressed by αC).

5.6 Multiple Inheritance

In the following example, we want to visualize the average salary (e.g., over the years) of people from different states. The average salary is visualized with the visual language `barChart`. In the case of RI, the "ocean state", we want to compose the visual language `barChart` with the visual language `blue`. The visual language `blue` is the language of all visual objects that have the color blue. We call the visualization composition operator, $\oplus$, and the new visual language is therefore `barChart`$\oplus$ `blue` (see Figure 19). The labelled refbox is presented in Figure 20. In the above example, both visual languages have the same attributes. Otherwise, if the value of an attribute is undefined for one visualization we say that its value is *undefined* and {*undefined*, X} = {X}. On the other hand, the attribute color could have been defined for both visual languages, in which case there would be the need to define what is meant by $color_1 \oplus color_2$ for different values of the color attribute. For example, $blue \oplus yellow$ can be defined to be *green*, and $red \oplus green$ can be defined to be *red*.

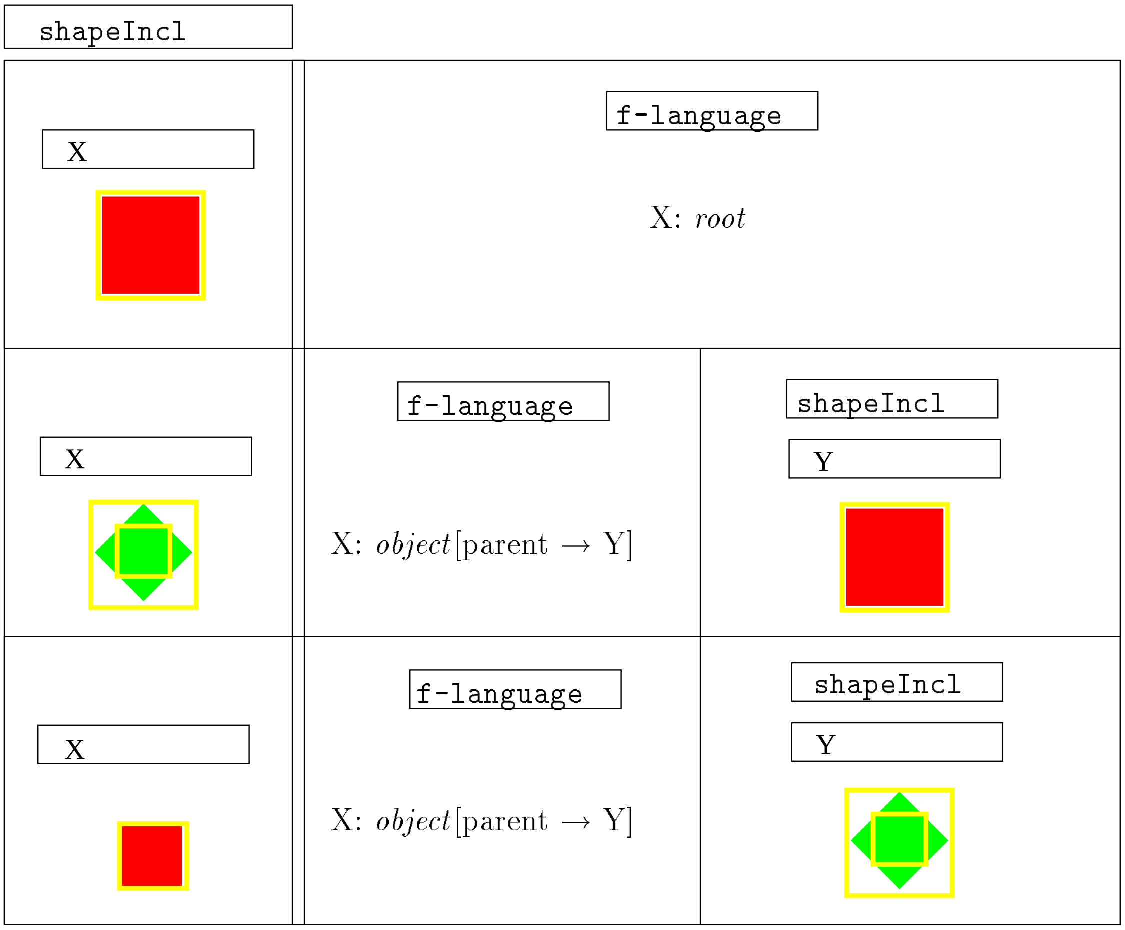


Figure 15: DOODLE program that performs visual recursion.

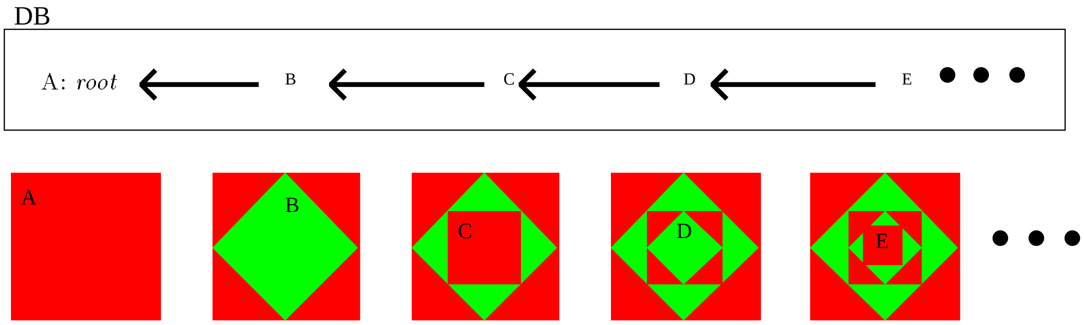


Figure 16: The pictures of `shapeIncl` obtained with the program of Figure 16 and the database DB.

5.7 Meta-queries

In DOODLE, a program that visually groups all the classes for which the visual language `pieChart` has been defined, is shown in Figure 21. For the visualization `pieChart`, this program specifies a rectangle inside of which nodes are placed. These nodes are labelled by the class names of that can be visualized with `pieChart` (this is represented by the string `‘C‘`.

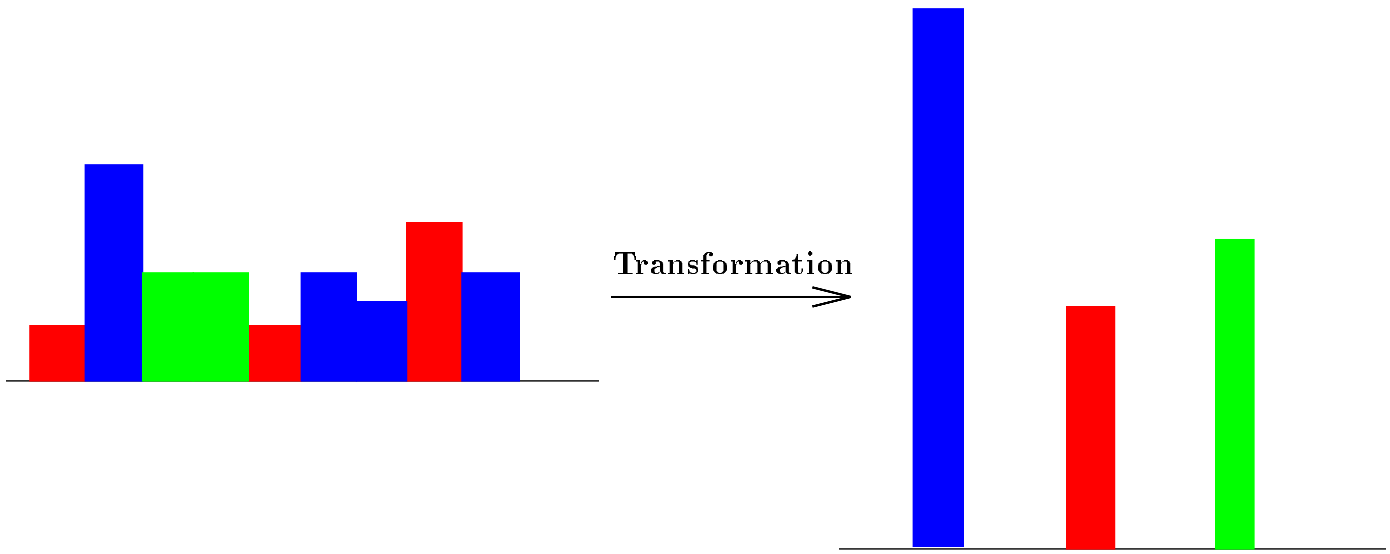


Figure 17: Transformation from a bar chart to a summarized bar chart.

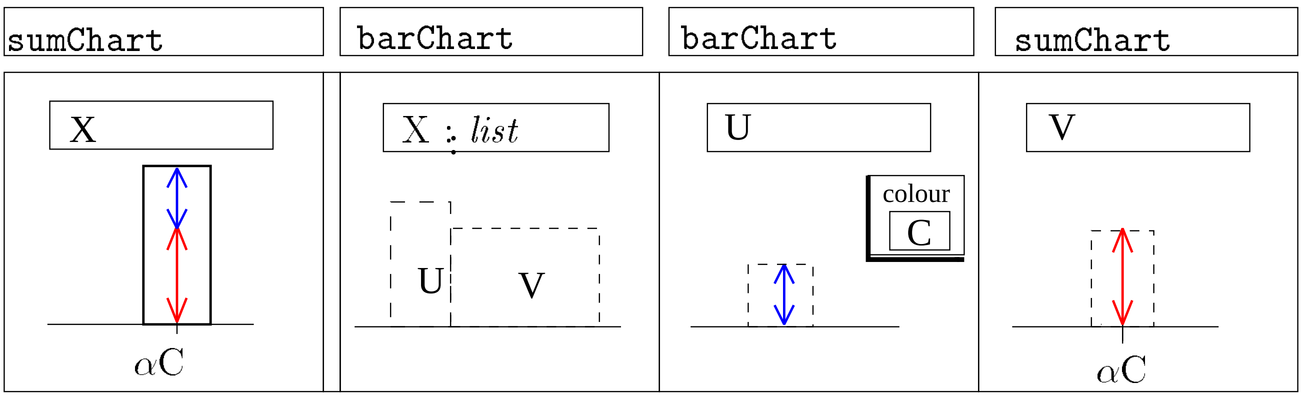


Figure 18: Visual program that defines `sumChart`.

6 Visual Expressiveness Issues

Mackinlay [Mac86] observes that visual languages have a limited expressiveness. A language may not be able to encode a given set of database facts, for two reasons: (1) certain facts are not expressible in the visual language; (2) a picture in the visual language may also express spurious facts, that is, facts that are *not* in the database. In this section we discuss expressiveness issues. In order to talk about pictures and the information they convey we define the *information content of a picture* to be the set of visual facts that are instances of the relevant predicates. The concept of a relevant predicate is useful because:

- Not all visual facts convey meaning in a picture (compare the predicate *above* in an unconstrained layout of a tree, with the same predicate in an downward drawing of a tree).
- Relevant predicates can also be used to model the expectations/capabilities of the user. For example the predicate right of (and left of) is important to the layout of Pert networks, when that layout is "read" left to right; the predicate *color* is not relevant when the available workstations are black and white.

Mackinlay [Mac86] claims that certain data visualizations may produce unsound pictures, that is, pictures that express relationships that are not true in the database. as exemplified by the visual transitive relationship *contains* when used to depict the non-transitive relationship *neighbor*, as in Figure 22. According to

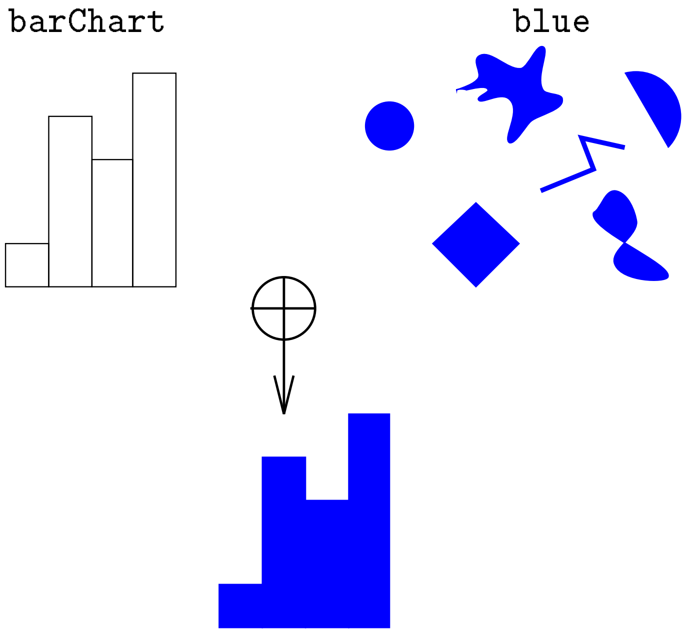
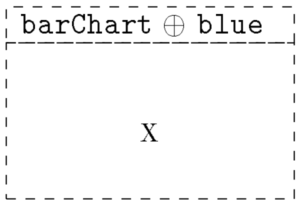


Figure 19: Composition of visual languages.



X: *avg_salary* [state → RI]

Figure 20: Labelled refbox.

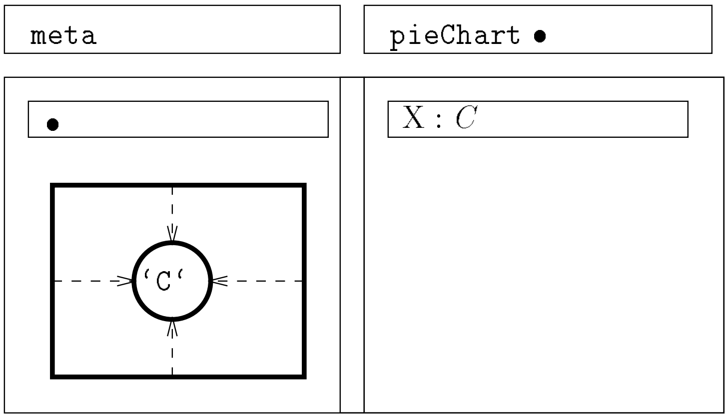


Figure 21: Example of a meta-query in DOODLE.

Mackinlay, the user would infer (wrongly) that Canada is a *neighbor* of Mexico, because the rectangle that depicts Canada *contains* the rectangle that depicts Mexico.

A logical framework for depiction and image interpretation is described in [RM89]. We extend this work with *graphical rules*, which express the user's perception of relevant predicates (e.g., that the visual predicate

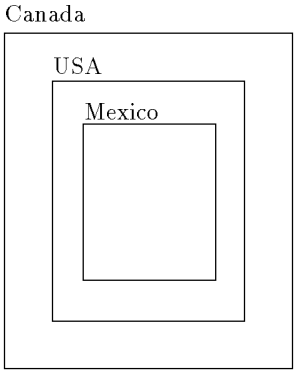


Figure 22: Visual representation that conveys unsound facts.

contains is transitive). This framework is also useful to account for the fact that not all facts need to be visualized explicitly, if the same information can be extracted implicitly. We can formally define soundness and completeness of a visual language with respect to a database, and we also define the stronger concept of adequacy. Given a visual language V, a set of relevant predicates, and a database schema, let T be the set of facts that are a logical consequence of the database facts, the graphical rules for the relevant predicates, and the visualization rules (i.e.,the rules that express the mapping from facts to visual objects), and let T' be the set of facts that are a logical consequence of the visual facts and of the interpretation rules. V is *sound* for the database schema, if for any database with that schema $T' \subset T$. V is *complete* for the database schema, if for any database with that schema $T \subset T'$. A visual language is *adequate* to visualize a database schema if it is both complete and sound, for any instance of the schema. As a consequence, box containment is *not* an adequate visual representation for a binary relation. This happens because it is known that not every partial order can be represented by the inclusion relation among boxes (see for example [Urr87]).

We also identify other kinds of mismatches between data and its visual representation. For example:

Ambiguous pictures. These are pictures that for the same relevant predicates and graphical rules, there is more than one set of visual facts that are entailed by the picture.

Pictures that encode "impossible" databases. An example includes expressing a visual predicate and its negation at the same time. This corresponds to an empty database model.

Given a set of graphical conventions it is not always possible to visualize a set of facts. For example, flights' durations may not respect the triangle inequality. This means that given a graph that depicts the physical location of the cities of arrival and departure, the duration of flights cannot be represented by straight lines between the cities, if the duration of the flights is to be proportional to the length of the lines.

In the case of transformations (i.e., mappings from visual languages to other visual languages) we call the visual language (or visual languages) we map from the *source language(s)*, and the resulting language we call the *target language*. We consider, similarly to the visualization rules, transformation rules. *Transformation rules* are logical rules that express the mapping from the source language(s) to the target language. In what follows we discuss picture equivalence, that is, when two pictures that are instances of different visual languages contain the "same information" about a set of database facts. We motivate this discussion with Figure 23. In Figures 23(i,ii) and 23(1,2) the transformation rules do not affect the relevant visual predicates, respectively *contains* and *connects*. For example, the transformation rule for Figure 23(i,ii) is

$$contains(\mathrm{X},\ \mathrm{Y}) \longleftarrow contains(\mathrm{X}',\ \mathrm{Y}'),\ trans(\mathrm{X}',\ \mathrm{X}),\ trans(\mathrm{Y}',\ \mathrm{Y}).$$

In this rule, the predicate *trans* encodes mappings between the graphical objects in the source language and the graphical objects in the target language.

$trans(a,\ a')$	*(graphical object a is mapped to graphical object a')*
$trans(b,\ b')$	*(graphical object b is mapped to graphical object b')*

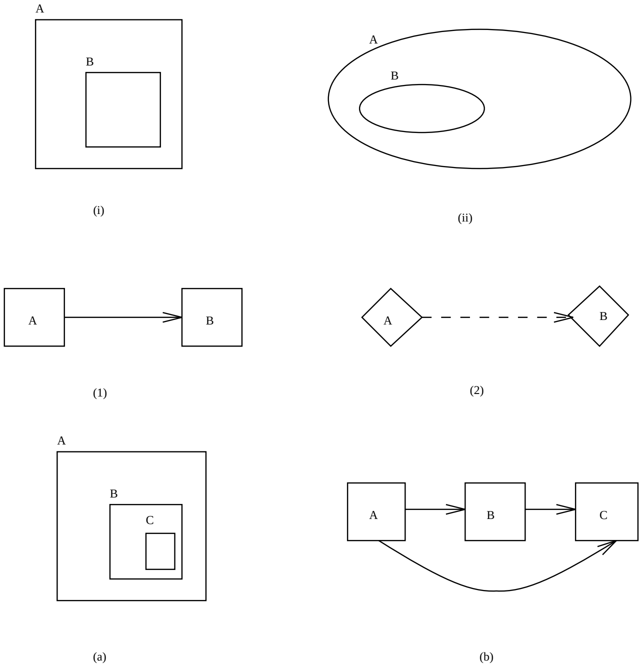


Figure 23: (i) and (ii) are equivalent under "inclusion" (or *preserve* inclusion). (1) and (2) are equivalent under "connection". Is there a measure for comparing (a) and (b)?

In addition, if a and a' depict the same database object, and b and b' depict the same database object, then it is easy to show that both pictures represent the same set of database facts. The pictures of Figures 23(a,b) do not preserve the visual predicate *contains*. In this case, to determine if the two pictures convey the same database information, we have to consider the graphical rules for the predicate *contains* and the graphical rules for the predicate *connects*, and compare the sets of database facts that are implied by both.

The fundamental concept of *genericity* in the relational model expresses the invariance of queries under the permutations of the elements of the domains [CH80, Hul86]. As transformations of visual languages preserve only certain predicates (see Figures 23(i), and (ii)), the notion of genericity for visual languages should consider only those permutations of the elements of the domain that preserve the relevant visual predicates.

We say that a query Q is *generic* if it satisfies the following *consistency criterion*.

$$\textit{If} \;\; B \overset{\Pi}{\leftrightarrow} B' \;\; \textit{then} \;\; Q(B') = \Pi(Q(B))$$

for any permutation Π of the values of some property B (such as shape or width) of the graphical objects that preserves the relevant visual predicates.

In Figure 24, we provide an example of genericity. The visual property whose values are being permuted by Π is "shape". Note that the permutation preserves the relevant visual predicate "connects". Query Q computes the transitive closure and visualizes it by means of containment.

7 A System's Architecture for DOODLE

Figure 25 depicts the building blocks of a system that supports DOODLE: The system blocks are represented in solid lines and the rules and objects that are produced by the system are enclosed in dashed boxes. The entities that are directly visualized by the user have the representation of an "eye" near them.

The *program assistant* supplies a menu of prototypical and key symbols for the user to compose a DOODLE program. In a multi-window environment, this program runs on one (or more) windows, which are dedicated to it.

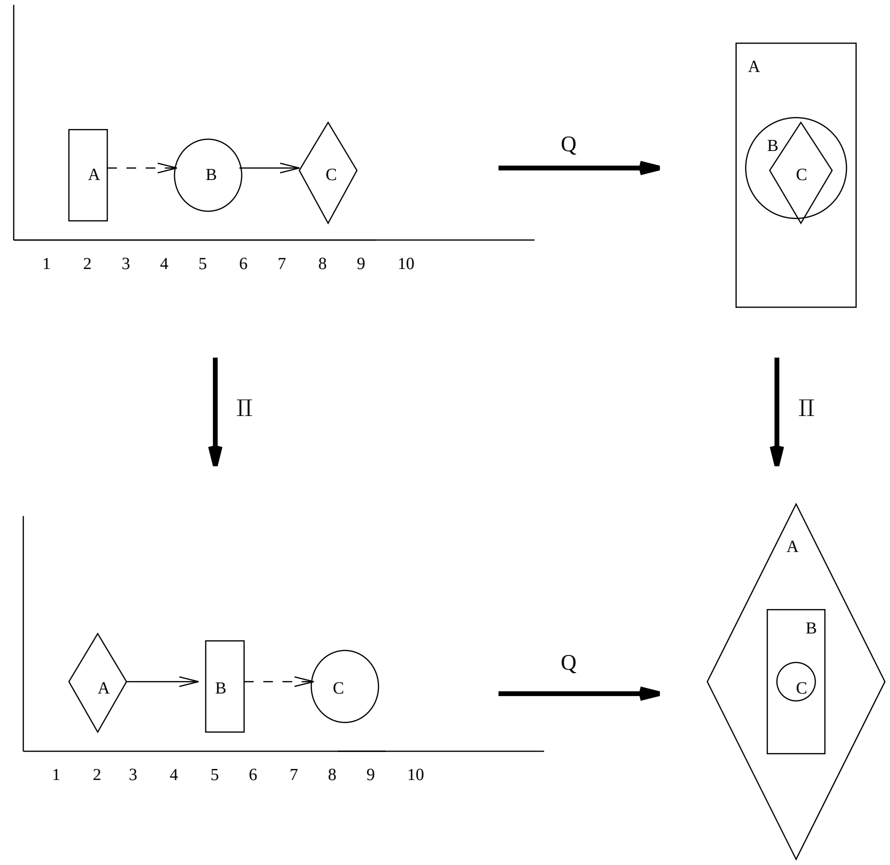


Figure 24: Example of genericity.

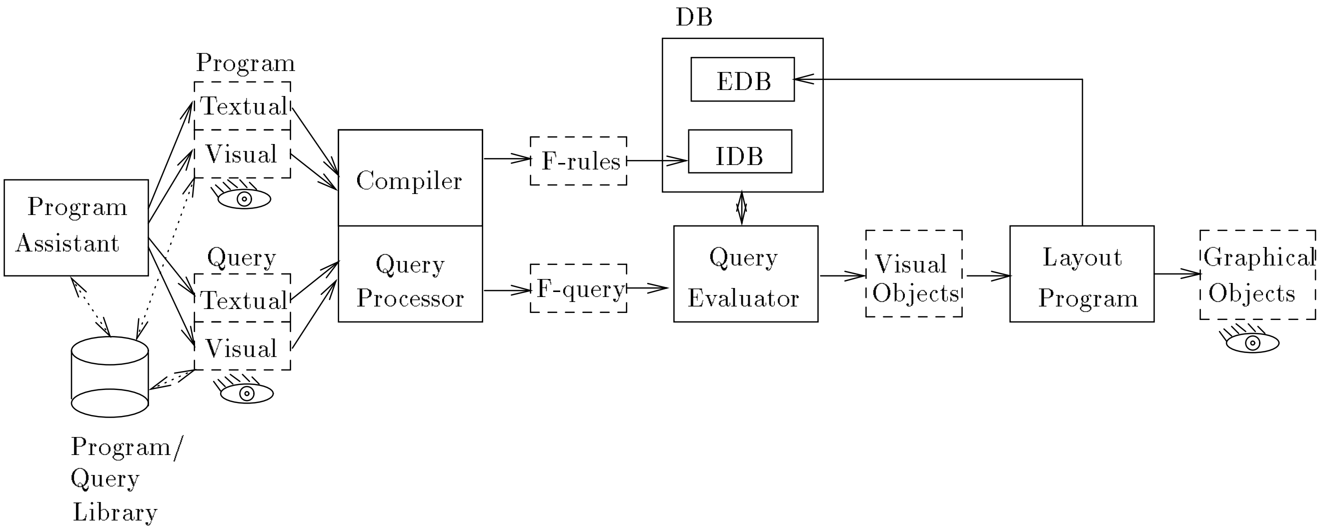


Figure 25: A system for DOODLE.

The user may compose a DOODLE program either from scratch using a textual and picture editor, or by loading another program and modifying it. Once a visual program (or query) is complete it gets processed by the compiler (or query processor), which transforms the DOODLE visual program into a F-logic program or query. The F-logic query is evaluated by the query evaluator which has access to the facts in the database and to the F-logic rules that encode the visual programs that are contained in the IDB (intensional database).

The visual objects (and visual constraint objects) specify the answer to the query. These objects are not necessarily ground terms, as illustrated by Figure 1.

The *layout program* creates pictures on the screen, with the (sometimes partial) information that is supplied by the visual objects. The textual representation of the visual and graphical objects are stored in the EDB, so that they can be queried in turn.

8 Conclusions and Future Work

In this paper we give examples of visual languages that can be expressed with DOODLE, and in particular we show that: (1) Visual languages are first-class citizens of the model: it is possible to *transform*, *compose*, and *inherit from* already defined visual languages to form new visual languages; (2) All visual operations on the data are semantically deductive and object-oriented queries. We also present a variety of applications of DOODLE, and outline the U-term language, which is a language for specifying constraints visually [Cru93].

As mentioned in the introduction, DOODLE is very different from other visualization and query languages, and motivates a variety of new research directions, such as:

Expressive Power. We plan to investigate the expressive power of DOODLE as a query language, and to extend DOODLE with visual constructs that express negation. Also adding non-determinism (see, e.g., [BK93]) would support, for example, the specification of mappings between sets and lists.

3D U-term Language. We plan to extend the U-term language to specify 3D visualizations [Rei93].

Visual Graph Drawing. We are currently investigating the integration of the algorithmic and declarative paradigms for graph drawing through visual constraints [CTV93].

Temporal Databases. We believe that temporal databases could benefit from an interface like DOODLE. Temporal charts (see for example [Tuf90]) could be used for expressing temporal queries.

Acknowledgements

Thanks to Michael Kifer, Yoram Kornatzky, Alberto Mendelzon, Theo Norvell, and Roberto Tamassia for useful discussions.

References

[Abi88] Serge Abiteboul. Updates, a New Frontier. In *Intl. Conference on Database Theory*, pages 1–18, 1988.

[AEM86] T. Lougenia Anderson, Earl F. Ecklund,Jr., and David Maier. PROTEUS: Objectifying the DBMS User Interface. In *Intl. Workshop on Object-Oriented Database Systems*, 1986.

[AK89] Serge Abiteboul and Paris C. Kanellakis. Object Identity as a Query Language Primitive. In *ACM-SIGMOD Intl. Conf. on Management of Data*, pages 159–173, 1989.

[AV87] Serge Abiteboul and Victor Vianu. Datalog Extensions for Database Queries and Updates. Technical Report 900, INRIA, 1987.

[Ber83] Jacques Bertin. *Semiology of Graphics*. The University of Wisconsin Press, Madison, Wisconsin, 1983.

[BK93] Anthony J. Bonner and Michael Kifer. Transaction Logic: An (Early) Exposé. In *Proc. of the Workshop on Formal Methods in Databases and Software Engineering*, 1993. Keynote address. Workshop held in Montreal, May 1992.

[Bor81] Alan Borning. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.

[CH80] Ashok K. Chandra and David Harel. Computable Queries for Relational Databases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.

[Cru90] Isabel F. Cruz. Declarative Query Languages for Object-Oriented Databases. In F. H. Lochovsky, editor, *Office and Data Base Systems Research '89*, pages 92–130. Technical Report CSRI-238, June 1990.

[Cru92] Isabel F. Cruz. DOODLE: A Visual Language for Object-Oriented Databases. In *ACM-SIGMOD Intl. Conf. on Management of Data*, pages 71–80, 1992.

[Cru93] Isabel F. Cruz. Expressing Constraints for Data Display Specification: A Visual Approach. Technical report, Dept. of Computer Science, Brown University, 1993. (also submitted for publication).

[CTV93] Isabel F. Cruz, Roberto Tamassia, and Pascal Van Hentenryck. A Visual Approach to Graph Drawing. In *Graph Drawing'93*, Sèvres, France, September 1993. `The abstracts of the papers are available via anonymous ftp from wilma.cs.brown.edu (128.148.33.66), in file gd93-v2.tex.Z in the directory /pub/papers/compgeo.`

[DETT93] G. Di Battista, P. Eades, R. Tamassia, and I.G. Tollis. Algorithms for Drawing Graphs: an Annotated Bibliography. Technical report, Department of Computer Science, Brown University, March 1993. To appear in *Computational Geometry: Theory and Applications.*

[GC90] T. C. Nicholas Graham and J. R. Cordy. GVL: A Graphical, Functional Language for the Specification of Output in Programming Languages. In *Proc. IEEE Intl. Conference on Computer Languages*, pages 11–22, 1990.

[Gol91] Eric J. Golin. A Method for the Specification and Parsing of Visual Languages. Technical Report CS-90-19, Brown University, May 1991.

[GPV90] Marc Gyssens, Jan Paredaens, and Dirk Van Gucht. A Graph-Oriented Object Database Model. In *ACM Symposium on Principles of Database Systems*, pages 417–424, 1990.

[GSKZ85] K.J. Goldman, S.A.Goldman, P.C. Kanellakis, and S.B. Zdonik. ISIS:Interface for a Semantic Information System. In *ACM-SIGMOD Intl. Conf. on Management of Data*, pages 328–342, 1985.

[GZG92] Sergio Greco, Carlo Zaniolo, and Sumit Ganguly. Greedy by Choice. In *ACM Symposium on Principles of Database Systems*, pages 105–113, 1992.

[Har88] David Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[Hul86] Richard Hull. Relative Information Capacity of Simple Relational Database Schemata. *SIAM J. Comput.*, 15(3):856–886, August 1986.

[Kam89] Tomihisa Kamada. *Visualizing Abstract Objects and Relations – A Constraint-Based Approach.* World Scientific, Singapore, 1989.

[KLW90] Michael Kifer, Georg Lausen, and James Wu. Logic Foundations of Object-Oriented and Frame-Based Languages. Technical Report 90/14 (2-nd revision), Department of Computer Science, SUNY Stony Brook, 1990. To appear in *JACM.*

[Mac86] Jock D. Mackinlay. Automatic Design of Graphical Presentations. Technical Report STAN-NCS-86-1138, Department of Computer Science, Stanford University, 1986.

[Rei90] Steven P. Reiss. Interacting with the FIELD Environment. *Software Practice and Experience*, 20(S1):89–115, 1990.

[Rei93] Steven P. Reiss. A Framework for Abstract 3D Visualizations. In *IEEE Symposium on Visual Languages*, pages 108–115, 1993.

[RM89] Raymond Reiter and Alan K. Mackworth. A Logical Framework for Depiction and Image Interpretation. *Artificial Intelligence*, 41:125–155, 1989.

[Tuf83] Edward R. Tufte. *The Visual Display of Quantitative Information.* Graphics Press., Cheshire, Conn., 1983.

[Tuf90] Edward R. Tufte. *Envisioning Information.* Graphics Press., Cheshire, Conn., 1990.

[Urr87] Jorge Urrutia. Partial Orders and Euclidean geometry. Technical Report TR-87-23, University of Ottawa, Department of Computer Science, September 1987.

[Wyk82] Christopher J. Van Wyk. A High-Level Language for Specifying Pictures. *ACM Transactions on Graphics*, 1(2):163–182, April 1982.