In: *13th IEEE Symp. on Visual Languages*, Capri, 1997. To appear.

Semantics of Visual Languages

Martin Erwig
FernUniversität Hagen
58084 Hagen, Germany
erwig@fernuni-hagen.de

Abstract

The effective use of visual languages requires a precise understanding of their meaning. Moreover, it is impossible to prove properties of visual languages like soundness of transformation rules or correctness results without having a formal language definition. Although this sounds obvious, it is surprising that only few work has been done about the semantics of visual languages, and even worse, there is no general framework available for the semantics specification of different visual languages. We present such a framework that is based on a rather general notion of abstract visual syntax. This framework allows a logical as well as a denotational approach to visual semantics, and it facilitates the formal reasoning about visual languages and their properties. We illustrate the concepts of the proposed approach by defining abstract syntax and semantics for the visual languages VEX, Show and Tell, and Euler Circles. For the latter we also prove a rule for visual reasoning.

1. Introduction

Investigating the semantics of visual languages is important for several reasons: First of all, a precise definition of semantics is indispensable for a thorough understanding of any language. This in turn is important to appraise a visual language and to compare it to others. Furthermore, this facilitates the development of extensions or a re-design of the language. Second, having a precise specification of a language's semantics, it is in many cases only a small step toward an implementation, for instance, denotational semantics can be translated almost verbatim into functional languages, so that an interpreter for the language is immediately available [12]. Third, with a precise semantics, various properties of languages can be proved. In particular, we can prove syntactic transformations to be sound w.r.t. the semantics (e.g., β-reduction in VEX can be shown to realize function application, or rules for syllogistic reasoning in Euler diagrams can be proved sound). Finally, a clear semantics of visual languages is needed to integrate them correctly into other environments. This especially applies to heterogeneous or multi-paradigm languages, see e.g. [7].

Despite the reasons just mentioned, research on visual language semantics is rather sporadic. In particular, there is no general framework available which could be used for the formal specification of visual languages. This situation is quite different than in textual languages: There we can choose among a variety of different semantic formalisms, such as denotational semantics, structured operational semantics, action semantics, evolving algebras etc., and some of these could, in principle, be employed for visual languages as well. A possible reason why this does not happen might be that some of the components that are necessary for a semantics framework are missing. Taking denotational semantics as an example, we observe that – at least as far as visual *programming* languages are concerned – the necessary concepts of semantic function and semantic domain can be used as in the textual case. However, the third component, *abstract syntax*, cannot be simply taken for visual languages, and there is no equivalent notion for visual languages yet.

So in the sequel we will first introduce a concept of abstract visual syntax in Section 2 before we demonstrate the specification of logical and denotational semantics in Sections 3 and 4 by two simple examples. In Section 5 we show that also more complex visual languages can be dealt with by the presented approach. Section 6 comments on related work, and Section 7 presents some conclusions.

2. What is Abstract Visual Syntax?

A textual language L is a set of strings over an alphabet A, i.e., $L \subseteq A^*$. The symbols of any sentence (or word) $w \in L$ are only related to each other by a linear ordering. In contrast, a sentence (or diagram or picture) p of a visual language VL over an alphabet A consists of a set of symbols of A that are, in general, related by several relationships $\{r_1, \ldots, r_n\} = R$. Thus we can say that a picture p is given by a pair (s, r) where $s \subseteq A$ is the set of symbols of the picture and $r \subseteq s \times R \times s$ gives the relationships that hold in p.[1] In other words, p is nothing but a directed graph with edge labels drawn from R, and a visual language is simply a set of such graphs.

Usually, languages contain a certain structure, i.e., there are precise rules defining which symbols can occur in which contexts and, regarding visual languages, which symbols may take part in which relationships. This structure is recognized and enforced during syntax analysis, and it can be assumed when defining semantics. Therefore, semantics definitions are often based on so-called *abstract syntax* which defines a language on a more abstract level

1. Relationships with arity > 2 can always be simulated by several binary relationships.

with less constraints than on the concrete level. This means that a description of concrete syntax must include every detail about the language whereas the abstract syntax can safely ignore all aspects that are not needed within the semantics definition.

At least for visual languages, we can actually distinguish different levels of "abstractness": First, we can abstract from concrete symbols and from geometric details, such as size and position of objects.[1] Second, we can ignore associativities that are used to resolve ambiguous situations during parsing and typings which restricts relationships to specific subsets of symbols. Finally, we can even forget about structural constraints; then a picture is just considered a directed labeled graph with possibly remaining restrictions w.r.t. node and edge labels. Thus we can define:

Definition 1. A *directed labeled multi-graph of type* (α, β) is a quintuple $G = (V, E, \iota, \nu, \varepsilon)$ consisting of a set of nodes V and a set of edges E where $\iota: E \rightarrow V \times V$ is a total mapping defining for each edge the nodes it connects. The mappings $\nu: V \rightarrow \alpha$ and $\varepsilon: E \rightarrow \beta$ define the node and edge labels.

For a graph g, $V(g)$ ($E(g)$) denotes the set of nodes (edges) of g. The successors of a node are denoted by $succ(v)$, i.e., $succ(v) = \{w \in V \mid \exists e \in V: \iota(e) = (v, w)\}$. Likewise, $pred(v)$ denotes v's predecessors.

The label types α and β might just be sets of symbols, or they can be complex structures to enable the labeling with terms, semantic values, or even graphs (see Section 5). The set of all graphs of type (α, β) is denoted by $\Gamma(\alpha, \beta)$.

Definition 2. A *visual language of type* (α, β) is a set of graphs $VL \subseteq \Gamma(\alpha, \beta)$.

In the sequel we will look at visual languages on this very abstract level, i.e., the abstract syntax of a visual language is specified as a set of graphs of a specific type.

How does this view relate to the well-established grammatical approach to syntax? Clearly, the syntax of languages can be conveniently specified by grammars. Grammars provide a way to generate all sentences of the language and, given a suitable parsing algorithm, allow to test whether a sentence is a member of the language (possibly giving a proof for this by constructing a parse tree for reconstructing the sentence). Concerning abstract syntax, however, grammars are usually not used for parsing; their purpose is just to offer an inductive or decompositional view of language that facilitates semantics definitions, especially, denotational semantics or structured operational semantics. As demonstrated in [6] we can actually have a (de)compositional/recursive view of graphs without resorting to grammars. So we can achieve a highly abstract comprehension of pictures together with an inductive view of graphs that facilitates, say, denotational semantics definitions. On the other hand, there are visual languages whose semantics are best described in a logical fashion. In that case a global, set-theoretic view of language is needed, which is just given by abstract visual syntax (and which might be obscured when using grammatical formalisms).

As in the textual case the choice of abstract syntax for a visual language is by no means unique. Usually, one has to trade similarity to the original notation for simplicity of the semantics definition. We will illustrate this point in Section 4.

1. At least up to "topological equivalence", i.e., as long as (relevant) relationships between objects are not affected.

3. Logical Semantics

In many cases, a logical specification of semantics views the syntactic elements simply as sets. For graphs, the node- and edge-set view is implicit in the definition. In Section 3.1 we define syntax and semantics of the well-known Euler diagrams, and in Section 3.2 we prove a visual rule for syllogistic reasoning and thus illustrate how to establish properties of a formalized visual language.

3.1 Euler Diagrams

The language of Euler diagrams as described in [8, 15] contains four kinds of basic pictures expressing logical statements:

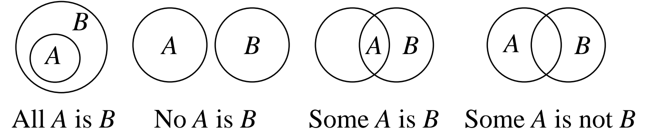


Fig. 1: Euler Diagrams

Ambiguities of Euler diagrams and semantic problems arising from these are discussed in detail in [15]. Our aim is not arguing in favor of or against using Euler diagrams for reasoning. However, as a matter of fact, Euler diagrams are a wide-spread visual notation, and in order to discuss the notation and compare it with others, it should be understood in the first place. This is what abstract visual syntax and the semantic formalism can achieve.

The concrete syntax of Euler diagrams comprises circles and string-labels together with the relationships *inside*, *intersects*, and *disjoint*. Labels have two purposes: First, they provide references to set symbols in pictures to be used in explanations, discussions etc. Second, their position distinguishes two different set relationships for intersecting circles. In the abstract syntax we can therefore omit labels and replace the *intersects*-relationship by two edge labels identifying the third and fourth situations, namely, *p-intersects* and *nic*. The names result from the following observations: In order to give a formal semantics to Euler diagrams one has to answer the following questions (among others): (1) Does the third situation also say: "Some B is not A" ? Yes, Euler also specifies that "Some A is not B" (and "Some B is A"). Thus we know: (i) $A \cap B \neq \emptyset$, (ii) $A - B \neq \emptyset$, and

(iii) $B - A \neq \emptyset$. So this situation describes what we call *proper intersection*, i.e., we say *A p-intersects B*. (2) Is the relative position of labels irrelevant, i.e., does the last example also say "Some *B* is not *A*" ? This would be reasonable, and although Euler gives as one possible instance an example where *B* is completely inside (i.e., properly included in) *A*, he himself uses the notation in a symmetric way later on in his letters. Accordingly, we ignore relative positions of labels. So this relationship describes that both differences are non-empty which expresses nothing but the fact that two sets are not comparable w.r.t. inclusion; we call this relationship *n*ot *i*nclusion-*c*omparable.

Except *inside*, all relationships are symmetric.We depict a symmetric relationship by an undirected edge which is represented in a directed graph by two directed edges in both directions. So the abstract syntax graphs for the Euler diagrams of Figure 1 look like:

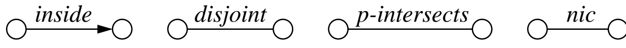


Fig. 2: Abstract Graphs for Euler Diagrams

The semantics is defined for a diagram relative to a *universe* of objects *U*. An interpretation is a mapping from the set of circles in the diagram, i.e., nodes of the graph, to subsets of *U*, i.e., $f: V \rightarrow 2^U$. Now the semantics can be easily defined:

$$S[(V, E)]U = \{f \mid f: V \rightarrow 2^U \wedge \forall e \in E: valid(f, \iota(e), \varepsilon(e))\}$$

where

$$valid(f, (u, v), l) = \begin{cases} f(u) \subseteq f(v) & \text{if } l = inside \\ f(u) \cap f(v) = \emptyset & \text{if } l = disjoint \\ f(u) \cap f(v) \neq \emptyset \wedge f(u) - f(v) \neq \emptyset \wedge f(v) - f(u) \neq \emptyset & \text{if } l = p\text{-}intersects \\ f(u) - f(v) \neq \emptyset \wedge f(v) - f(u) \neq \emptyset & \text{if } l = nic \end{cases}$$

3.2 Soundness of Visual Reasoning Rules

Having a precise definition of what Euler diagrams mean it is quite easy to check the visual rules for syllogistic reasoning. Euler gives textual versions of such rules and explains them by pictures. One example is:

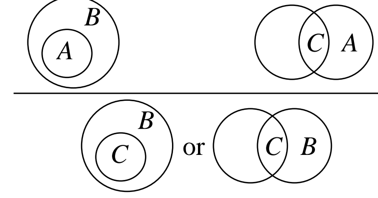
$$\frac{\text{All } A \text{ is } B \qquad \text{Some } C \text{ is } A}{\text{Some } C \text{ is } B}$$

Although this sounds very intuitive, this rule is formally *not* correct since "Some *C* is *B*" does only hold if $C - B \neq \emptyset$. But this cannot be concluded from the premises; *C* might well be included in *B*. Actually, Euler is aware of this fact and gives pictures illustrating both cases. The point is that there is no formal correspondence between propositions and pictures (since there is no formal semantics). Now the correct rule is:

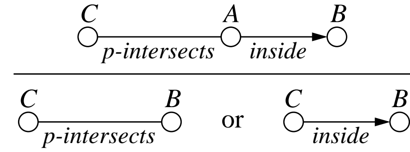
$$\frac{\text{All } A \text{ is } B \qquad \text{Some } C \text{ is } A}{\text{All } C \text{ is } B \text{ or Some } C \text{ is } B}$$

or equivalently in visual terms:

Lemma 1.



Proof. We reformulate this rule in terms of abstract syntax. The premises can be joined into one graph.



The semantics definition ensures for each valid interpretation the following properties:

(1) $A \subseteq B$ (2) $A \cap C \neq \emptyset$ (3) $A - C \neq \emptyset$ (4) $C - A \neq \emptyset$

First we observe from (3) and (4) that neither *A* nor *C* is empty. By (1) it also follows that *B* is not empty. For the intersection and difference of two non-empty sets we know:

(i) $X \cap Y \neq \emptyset \Leftrightarrow \exists Z \neq \emptyset: Z \subseteq X \wedge Z \subseteq Y$
(ii) $X - Y \neq \emptyset \Leftrightarrow \exists Z \neq \emptyset: Z \subseteq X \wedge Z \cap Y = \emptyset$

Next we translate the conclusion of the rule into formal terms. That is, have to show that the following is true:

$$(C \cap B \neq \emptyset \wedge C - B \neq \emptyset \wedge B - C \neq \emptyset) \vee C \subseteq B$$

We can simplify this term: First, since $C \subseteq B$ implies $C \cap B \neq \emptyset$, we have $C \cap B \neq \emptyset \vee C \subseteq B = C \cap B \neq \emptyset$, and second, $C - B \neq \emptyset \vee C \subseteq B$ is always true which can be easily checked by considering all possibilities w.r.t. to the intersection of *C* and *B*. Thus it remains to be shown:

$$C \cap B \neq \emptyset \wedge (B - C \neq \emptyset \vee C \subseteq B)$$

We can prove both parts separately. First, from (2) and (i) we infer $\exists D \neq \emptyset$: (5) $D \subseteq A$ and (6) $D \subseteq C$. By transitivity it follows from (5) and (1) that $D \subseteq B$, and this together with (6) and (i) implies $C \cap B \neq \emptyset$. Second, we obtain from (3) and (ii) the relationships $\exists D \neq \emptyset$: (7) $D \subseteq A$ and (8) $D \cap C = \emptyset$. By transitivity it follows from (7) and (1) that $D \subseteq B$, and this together with (8) and (ii) implies $B - C \neq \emptyset$. This means that $B - C \neq \emptyset \vee C \subseteq B$ is also true. □

4. Recursive Semantics

In contrast to the predicative view that was convenient in the previous section, many languages are defined inductively, and then a semantics definition is easiest to give when adopting that inductive view. We illustrate these ideas

with the visual language VEX [4], which provides a visual notation for the lambda calculus. We chose VEX, since it is a rather small (but computationally complete) language and since any semantics can be easily verified by comparison with the classical lambda-calculus.

In Section 4.1 we explain VEX informally, followed in Section 4.2 by two alternative abstract syntax definitions. Sections 4.3 and 4.4 introduce an inductive/decompositional view of syntax graphs that is particularly needed for the definition of denotational semantics. Based on this, a semantics for VEX is then given in Section 4.5.

4.1 Example: VEX

VEX [4] is a purely[1] visual language: Each identifier is represented by an (empty) circle that is connected by a straight line to a so-called *root node*. A root node is again an empty circle with one or more straight lines touching it, leading to all identifiers with the same name. A root node might be internally tangent to another circle, it then represents a parameter of an abstraction, otherwise it denotes a free variable. An abstraction has, in addition to its parameter circle, a body expression inside it. An application of two expressions is depicted by two externally tangent circles with an arrow at the tangent point. The head of the arrow lies inside the argument, and the tail of the arrow lies inside the abstraction to be applied. Application order can be controlled by labeling arrows with priority numbers which we will ignore for simplicity.

Figure 3 shows the VEX expressions for $(\lambda x.x)y$ and $\lambda y.((\lambda x.yx)z)$.

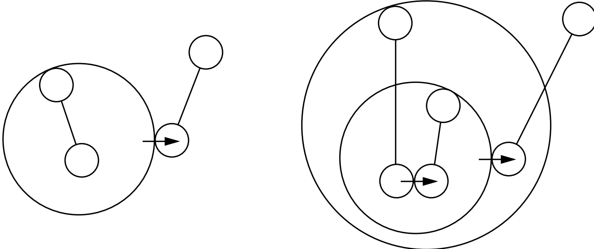


Fig. 3: Two VEX Expressions

Now what is the exact meaning of the above drawings? In [4] graphical rewrite rules are given that can be used to reduce VEX pictures to normal forms. This is, however, a pure syntactical manipulation. A true semantics definition maps VEX into a semantic domain of functions. In any case, the first step is a definition of abstract visual syntax for VEX.

4.2 Choices of Abstract Syntax

The VEX concrete syntax consists of symbols like circles, lines, and arrows, and relationships like inside or touches.

As already mentioned, there are quite different possibilities for the abstract syntax. In a first approach we can abstract from lines and arrows and replace them by corresponding relationships since lines simply link the use of a variable to its definition and arrows just indicate the application of one circle to another. This is reflected in the abstract syntax graph of a VEX expression by *def*-edges (i.e., edges labeled with *def*) that lead from a variable use to its definition and by *apply*-edges leading from the expression circle to be applied toward the argument circle. It remains to represent abstractions. An abstractions is given by a non-empty circle c where an (empty) circle x that is internally tangent to c represents c's parameter and all other circles $e_1, \ldots, e_n$ inside c define the abstraction body. In the abstract syntax we represent this information by a *par*-edge from c to x and by *body*-edges $(c, e_1), \ldots, (c, e_n)$. Note that we do not need to distinguish abstraction nodes from variable nodes by an explicit label since the difference can always be told by looking at the incident edges – by this the abstract syntax is more similar to the concrete syntax. Therefore we do not use any node labels, and thus the abstract syntax for VEX is given by graphs of type $(\varnothing, \{def, apply, par, body\})$.

Figure 4 gives the abstract syntax graphs for the VEX pictures from Figure 3.

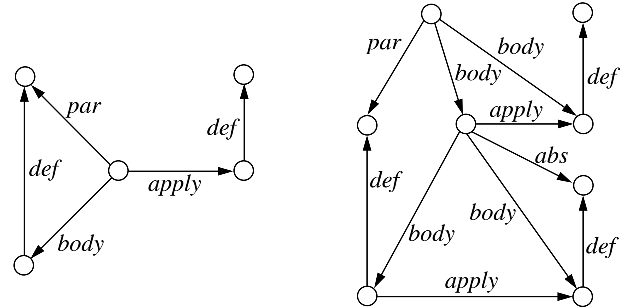


Fig. 4: Abstract Graphs for VEX Expressions

This representation is rather close to the spatial original and should therefore be easy to grasp. However, the semantics definition gets a bit involved, and defining β-reduction on the basis of this representation is quite difficult. In contrast, a DAG representing the lambda-expression in a rather traditional way allows a rather straightforward denotational semantics definition.

Such a representation consists of application-, abstraction- and variable-nodes (with corresponding node labels: @, λ, ○).[2] An @-node has an outgoing *fun*-edge and an outgoing *arg*-edge that lead to the function to be applied and the argument, respectively. A λ-node is connected by

1. Labels are sometimes used for illustration, but strictly, they are not needed.

2. Note that we do not need node labels to distinguish variables. As in the previous approach, uses of variables are linked by edges to the corresponding definitions. This mechanism is a perfect substitute for the "equal name"-method of the textual lambda-calculus. Therefore, nodes representing variables are left unlabeled.

an outgoing *par*-edge to its parameter, an unlabeled node, and by an outgoing *body*-edge to the node representing its body. Hence, this abstract syntax for VEX uses graphs of type ({@, λ}, {*fun*, *arg*, *par*, *body*}).

Figure 5 shows the abstract syntax graphs that correspond to the VEX pictures of Figure 3.

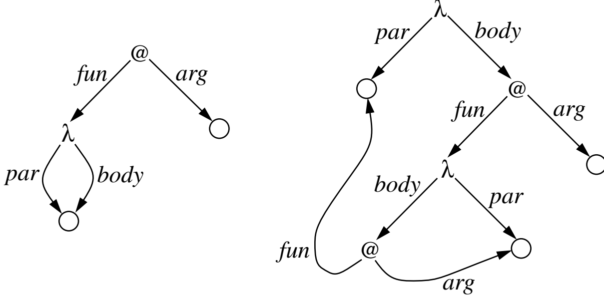
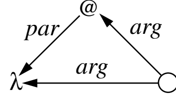


Fig. 5: Alternative Abstract Graphs for VEX

At this point it is important to recall that the informally stated structural properties are not captured by abstract syntax graphs. This means that a graph like that shown below is also a graph of the above type although it is certainly not representing any VEX expression For defining semantics we can safely assume structurally correct graphs be delivered, say, by a syntax analysis phase or an editor. The structural assumptions can then appear implicit in the semantics definition since we need only give semantics for structurally well-formed graphs, i.e., syntactically correct pictures.



Although the second representation offers advantages in the semantics definition, it does only poorly reflect the visual structure of the VEX expression, and might thereby complicate the understanding of the original *visual* language. The decision of which representation to chose depends on what is done with the semantics definition: For just giving a meaning to VEX pictures, the first approach might be sufficient, however, when trying to prove, e.g., soundness of β-reduction, or deriving an implementation, the second representation would probably be favored.

Next we would like to define the semantics on the basis of the abstract representations just given. We therefore need a structured way of accessing all the elements of a syntax graph. In particular, we need an inductive view of graphs that allows the structured, step-by-step decomposition of graphs. We will address this issue in the next two subsections. The concepts presented there can also be used to map between different syntax representations.

4.3 An Inductive Graph Model

We can view a graph in the style of algebraic data types found in functional languages like ML or Haskell: A graph is either empty, or it is constructed by a graph *g* and a new node *v* together with edges from *v* to its successors in *g* and edges from its predecessors in *g* leading to *v*. This way we can construct graphs expressions with a constant constructor *Empty* and a constructor *N* taking as arguments a triple (*pred-spec*, *node-spec*, *succ-spec*), called *node context*, and the graph *g* to be extended. Here, *node-spec* is a node identifier not already contained in *g* possibly followed by a label (e.g., *d*:@) and *pred-spec* (*succ-spec*) denotes a list[1] of predecessor (successor) nodes possibly extended by labels for the edges that come from (lead to) the nodes. E.g., [*d*›*fun*, *e*] denotes a list of two predecessor nodes *d* and *e* where the edge coming from *d* has label *fun* and the edge coming from *e* has no label at all. Similarly, [*par*›*a*, *body*›*a*] denotes a single successor *a* that is reached via two differently labeled edges.

E.g., the first graph from Figure 5 is given by the following expression:

N ([], *d*:@, [*fun*›*b*, *arg*›*c*]) (*N* ([], *c*, [])
(*N* ([], *b*:λ, [*par*›*a*, *body*›*a*]) (*N* ([], *a*, []) *Empty*)))

In the sequel we make use of two abbreviations: (1) empty sequences can be omitted, and (2) a cascade of *N*-constructors is replaced by a single *N**-constructor. So the above term can be simplified to:

*N** (*d*:@, [*fun*›*b*, *arg*›*c*]) (*c*)
(*b*:λ, [*par*›*a*, *body*›*a*]) (*a*) *Empty*

Note that there are, in general, many different graph expressions denoting the same graph. E.g., the above term denotes the same graph as:

*N** ([*d*›*fun*], *b*:λ, [*par*›*a*, *body*›*a*])
(*d*:@, [*arg*›*c*]) (*c*) (*a*) *Empty*

The following result is important since it guarantees that any graph can be viewed inductively:

Theorem 1. *Any directed labeled multi-graph can be represented by a graph expression.* □

The proof is not difficult, see [6]. There we also give a formal semantics of graph types and graph constructors.

4.4 Pattern Matching on Graphs

The main use of graph constructors in the context of this paper is not to build new graphs but to take part in pattern matching on graphs. Especially useful for graphs is the concept of *active patterns* [5]: Usually, matching a pattern like *N* (*p*, *v*:*l*, *s*) *g* to a graph expression binds the node context inserted last to *p*, *v*, *l*, *s* and the remaining graph to *g*. However, in order to move in a controlled way through the graph, it is necessary to match the context of a specific node. This is possible if *v* is already bound to the node to be matched. Then the context of *v* is bound to the remaining

1. Lists offer a convenient way for dealing with multiple edges between two nodes. In this respect, bags would also be fine, but lists can be sorted which eases the processing of, e.g., successors, in a specific order.

variables. E.g., matching the pattern $N\ (p,\ b{:}l,\ s)\ g$ against either graph expression from the previous subsection results in the following bindings:

$$p \rightarrow [d],\ l \rightarrow \lambda,\ s \rightarrow [a, a],\ g \rightarrow \text{“}\textit{rest-graph}\text{”}$$

where *rest-graph* is an arbitrary representation of the matched graph without node b and its incident edges (e.g., $N^*\ (d{:}@,\ [arg{\succ}c])\ (c{:}v)\ (a{:}v)\ Empty$).

We can restrict patterns further by adding labels that must be present or by replacing list variables by lists of a specific length. We can also ignore bindings by simply omitting the corresponding parts of the pattern. E.g., we can match the abstraction node b binding the parameter/body node to p/e by using the pattern: $N\ (b{:}\lambda,\ [par{\succ}p,\ body{\succ}e])\ g$. Actually, p and e will be bound to the same node, a. Since we did not specify anything for the predecessor list, no binding will be produced. If we wanted to ensure that the matched node has no predecessors we would have used the pattern $N\ ([],\ b{:}\lambda,\ [par{\succ}p,\ body{\succ}e])\ g$ instead. This, however, fails to match our example graph.

4.5 Denotational Semantics

Now we can define the denotational semantics of VEX. We map each syntax graph of a (syntactically correct) VEX expression into a value of a suitable domain $\boldsymbol{D}$ for the lambda-calculus (e.g., Scott's construction D_∞ or Plotkin's graph model $P\omega$ [2]). Let $\boldsymbol{d}$ be a variable denoting values from $\boldsymbol{D}$. It is interesting to note that in contrast to the denotational semantics of the textual lambda-calculus we do not need any environment for passing around variable bindings; we can rather employ the VEX root nodes to carry semantic values.

We define the semantics by moving in a controlled way through the abstract graph, i.e., semantics are given w.r.t. specific node contexts in the graph, and in the recursive definitions for the semantics of, say, node v, the semantics function S' is applied to the contexts of v's successors. Hence, S' has two parameters: a graph and a node determining the context. Using the second proposal for abstract syntax we can distinguish the following cases: first, the semantics of a node carrying a semantic value is the value itself. (Such a value is assigned by the rule for abstractions.) Second, the meaning of an application node is given by applying the semantics of the node connected by the *fun*-edge, which is expected to be a function value, to the value denoted by the argument node. Finally, the semantics of an abstraction is defined to be a function value (Λ denotes the semantic abstraction function) which maps any value $\boldsymbol{d}$ to the value denoted by the body of the abstraction when the parameter node is labeled $\boldsymbol{d}$. Note that in order to change the label of the parameter node p to $\boldsymbol{d}$ we have to decompose p from the graph and re-insert it with the new label and the old context (i.e., with predecessors r and no successors).

$$S'[v,\ N\ (v{:}\boldsymbol{d})\ g] = \boldsymbol{d}$$
$$S'[v,\ N\ (v{:}@,\ [fun{\succ}f,\ arg{\succ}a])\ g] = S'[f,\ g]\ (S'[a,\ g])$$
$$S'[v,\ N^*\ (v{:}\lambda,\ [par{\succ}p,\ body{\succ}b])\ (r,\ p)\ g] = \Lambda\boldsymbol{d}.S'[b,\ N\ (r,\ p{:}\boldsymbol{d},\ [])\ g]$$

Now the semantics of a graph g representing a VEX expression is given by applying S' to the root of g.

$$root(g) = \{v \in V(g) \mid pred(v) = []\}$$
$$S[g] = S'[the(root(g)),\ g]$$

Here, the function *the* simply extracts the one element from a singleton set and is undefined otherwise: $the(\{x\}) = x$.

5. A More Complex Example

In this section we consider abstract syntax and semantics of a more complex visual language: Show and Tell. The language is interesting for two reasons: first, it is a member of the rather large class of *data flow* languages and thus indicates how semantics could be defined for many other visual languages. Second, it demonstrates the effective use of nested syntax graphs which goes beyond grammatical descriptions of visual languages.

Show and Tell (STL) [11, 10] combines data flow with the concept of *completion*, which means to fill in empty boxes in a data flow graph by either computation or database search. Computations are represented by so-called *box-graphs*, which are acyclic directed multi-graphs whose nodes are rectangles connected by arrows. A box is empty or it contains either simple data, such as numbers or functions, or another whole box-graph. In that case the box is called *complex* and can be either *closed* or *open*. Data can flow along the arrows from one box to another. Whenever two boxes connected by an arrow contain different values, the box-graph is said to be *inconsistent*. An open box containing an inconsistent box-graph propagates this inconsistency, i.e., the box-graph containing the inconsistent box also becomes inconsistent. In contrast, when a closed box gets inconsistent, all that happens is that the box cannot receive or propagate any values, i.e., an inconsistent closed box can be viewed as deleted. With the concept of inconsistency, conditionals can be expressed without having boolean values.

Figure 6 shows an STL program implementing the logical AND. The program contains two parameters (the two topmost empty boxes) and one result (the empty box on the left). If both arguments are “1”, then the upper complex box remains consistent, and the “1” can flow directly into the result box. Moreover, the lower complex box gets inconsistent and cannot emit the “0”. On the other hand, if one argument is “0”, then the upper complex box gets inconsistent and cannot send data to the result box and to the lower box. Then, the “0” can flow from the lower box into the result box.

The abstract syntax mainly follows the concrete syntax

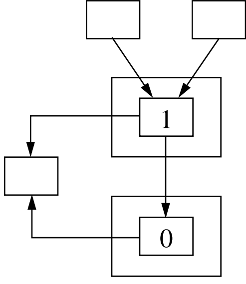


Fig. 6: STL program

In particular: (1) Nodes are labeled by constants (e.g., integers), function symbols (such as +), ○ (representing empty STL boxes), and complete graphs. Additionally, they carry an *open*- or *closed*-tag. (In the following we will mention these tags only when needed.) (2) Edges are labeled by pairs (i, j) where i means that the edge contributes to the ith parameter of the target node and j says that the jth component of the value at the edge's source node flows via this edge (here * means the complete value). (3) Each edge $e = (v, (i, j), w)$ (i.e., from v to w with label (i, j)) that crosses a border of a complex box u is replaced by a new node x with label k (lying inside u) and two edges e_1 and e_2 as follows: (i) If w is inside u, then $e_1 = (v, (k, j), u)$ (ending at u) and $e_2 = (x, (i, *), w)$ (connecting x to the target of e). (ii) If v is inside u, then $e_1 = (v, (1, j), x)$ and $e_2 = (u, (i, k), w)$. Here, k ranges from 1 to n (m) for all n incoming (m outgoing) edges. (4) The (top-level) box-graph is extended according to rule (2) as if it were enclosed by a box having edges ending at the roots and leaving the sinks.

The abstract syntax of the STL program from Figure 6 is show in Figure 7. Nodes with constants as labels are enclosed with circles and can thus be distinguished from newly introduced nodes.

If OP is the set of constants and operations used by STL programs, then STL abstract graphs without complex boxes have type $\Gamma(\alpha_0, \beta)$ with:

$$\alpha_0 = (OP \cup \{\bigcirc\} \cup \mathbb{N}) \times \{open, closed\}$$
$$\beta = \mathbb{N} \times (\mathbb{N} \cup \{*\})$$

Since complex boxes are represented by nodes labeled with abstract STL graphs, the node type can be inductively defined to include graphs of increasing nesting:

$$\alpha_{i+1} = \alpha_i \cup \Gamma(\alpha_i, \beta)$$

Hence, the type of arbitrary STL abstract graphs is given by $\Gamma = \cup_{i \geq 0} \Gamma(\alpha_i, \beta)$.

We can now define the semantics of each STL DAG as a function $\boldsymbol{D}^n \rightarrow \boldsymbol{D}^m$ when we take a domain of semantic values $\boldsymbol{D}$ (e.g., for integers) and add to it a special value ◊ for dealing with inconsistency (see below). The first equation selects all roots of the graph, assigns $\boldsymbol{D}$-variables as new labels, and yields a function over these variables:

$$S'[N^* ([], v_1{:}1, s_1) \ldots ([], v_n{:}n, s_n)\ g] =$$
$$\Lambda(\boldsymbol{d}_1, \ldots, \boldsymbol{d}_n).S'[N^* ([], v_1{:}\boldsymbol{d}_1, s_1) \ldots ([], v_n{:}\boldsymbol{d}_n, s_n)\ g]$$

The used cascade pattern with the ellipsis extends as far as possible, i.e., it selects all nodes labeled by integers and having no predecessors. The recursive application of S' denotes the result tuple (by applying another semantic function S'' to all sinks of the graph) together with the consistency status of the whole graph given by C.

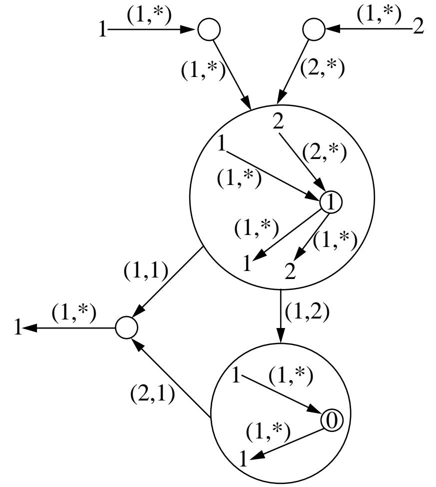


Fig. 7: Abstract Syntax of STL program

$$S'[N^* ([p_1], v_1{:}1, []) \ldots ([p_m], v_m{:}m, [])\ g] =$$
$$((S''[p_1, g], \ldots, S''[p_m, g]), C[g])$$

(Note that by definition of abstract syntax each sink has exactly one predecessor.) S'' moves in reverse direction through the abstract graph: It recursively determines the tuple of values for all predecessors and applies the function denoted by the current node to it. This function is denoted by the semantic function F given below. In the pattern we assume that the predecessors (p_i) are ordered w.r.t. the first label component (i) of the connecting edges. This ensures that the parameters appear in the correct order. Note that the values of the predecessors are not taken as a whole, but only the specific components as specified by the second label part (s_i) of the connecting edges. This is achieved by the application of projecting functions Πs_i (where $\Pi^*(x) = x$).

$$S''[v, N([p_1 \rangle (1, s_1), \ldots, p_k \rangle (k, s_k)], v{:}f)\ g] =$$
$$F[f]\ (\Pi s_1(S''[p_1, g]), \ldots, \Pi s_k(S''[p_k, g]))$$

The semantic functions S' and S'' only define the meaning of consistent STL-graphs. An inconsistent node or graph is defined to return the value ◊ which is equal to all other values of $\boldsymbol{D}$. (In this way, an inconsistent (closed) node that is connected by an edge to a node v that is labeled by a constant or not labeled at all does not affect the result of v.) A graph is inconsistent if any of its open nodes is inconsistent. Let *open* be a predicate that is true only for open nodes. The consistency of nodes/graphs is denoted by C'/C:

$$C'[v, g] = (open(v) \Rightarrow S''[v, g] \neq \lozenge)$$
$$C[g] = \forall v \in V(g){:}\ C'[v, g]$$

Now the semantics of an STL graph is simply given by:

$$S[g] = \begin{cases} \Pi_1(S'[g]) & \text{if } \Pi_2(S'[g]) \\ \Diamond & \text{otherwise} \end{cases}$$

It remains to define the functions denoted by node labels. An operations on $\boldsymbol{D}$ (like +) denotes itself. A constant c is interpreted as a function that checks whether all incoming values are equal to c, and an unlabeled node checks all incoming values for equality. Finally, the semantics of a node labeled by a complete STL graph is given by S.

$F[f : \boldsymbol{D}^n \rightarrow \boldsymbol{D}^m] = f$
$F[c : \boldsymbol{D}] = \Lambda(\boldsymbol{d}_1, \ldots, \boldsymbol{d}_n).\textit{if } \boldsymbol{d}_1 = \ldots = \boldsymbol{d}_n = c \textit{ then } c \textit{ else } \Diamond$
$F[\bigcirc] = \Lambda(\boldsymbol{d}_1, \ldots, \boldsymbol{d}_n).\textit{if } \boldsymbol{d}_1 = \ldots = \boldsymbol{d}_n \textit{ then } \boldsymbol{d}_1 \textit{ else } \Diamond$
$F[g : \Gamma] = S[g]$

6. Related Work

Besides semantics definitions for specific languages, such as [10], there is only few work dealing with semantics of visual languages in general. Wang and Lee [16] take an algebraic view of modeling pictures. Their goal is to get a formal basis for visual reasoning by axiomatic characterizations of what can be seen in a picture. The work of Bottoni et al. [3] is centered around the formal understanding of and reasoning with images. Both approaches are based on concrete visual syntax and are not targeted at the semantics specification of visual *programming* languages.

There is also some work related to the use of graphs for describing pictures: Harel's higraphs [9] are a kind of amalgam of hierarchical graphs and Euler/Venn diagrams. Higraphs have a concise formal semantics, and by modeling a visual language *VL* as a higraph, the semantics of VL is implicitly defined. Although quite many applications can, in principal, be described as higraphs, several of them require changes of their concrete syntax, and some languages cannot be described at all. Moreover, the lack of an inductive view of higraphs makes denotational specifications difficult, if not impossible. Graph grammars, on the other hand, provide an inductive view of graphs, but they have not yet been used for the specification of visual language semantics, rather they have been employed to describe translations. One reason could be that graph grammars and graph rewriting has a non-trivial semantics itself (caused by complex embeddings and non-determinism), and it is thus difficult to describe semantics on the basis of this formalism. Another difficulty is that graphs are considered as global, imperative variables (they are not parameters of grammar rules). In particular, this makes nesting of graph structures as used in the syntax for STL impossible.

Concerning abstract visual syntax, some other authors also recommend the separation from concrete syntax [1, 13, 14]. However, this is only partially achieved by those approaches, since they require a one-to-one correspondence between concrete and abstract syntax, and thus abstract syntax is intrinsically coupled very closely to concrete syntax. Also, that work is concerned with translation of visual languages, semantics are not covered.

7. Conclusions

We have presented a general framework for the specification of visual language semantics. A rather unrestricted form of abstract visual syntax given by graphs is the backbone of the formalism. The approach applies to quite a wide range of visual languages, and we can even employ different semantics formalism, such as denotational or logical semantics. Currently, we are using the framework to formalize other visual languages, and we investigate the automatic generation of compiler backends from semantics specifications.

8. References

[1] Andries, M., Engels, G. & Rekers, J.: How to Represent a Visual Program?, *Workshop on Theory of Visual Languages*, 1996.

[2] Barendregt, H.P.: *The Lambda Calculus – Its Syntax and Semantics*, North Holland, 1981.

[3] Bottoni, P., Costabile, M.F., Levialdi, S. & Mussio, P.: Formalising Visual Languages, *IEEE Symp. on Visual Languages*, 1995, 45-52.

[4] Citrin, W., Hall, R. & Zorn, B.: Programming with Visual Expressions, *IEEE Symp. on Visual Languages*, 1995, 294-301.

[5] Erwig, M.: Active Patterns, *Int. Workshop on Implementation of Functional Languages*, 1996, 95-112. To appear in LNCS.
ftp://ftp.fernuni-hagen.de/pub/fachb/inf/pri4/papers/ap.ps.gz

[6] Erwig, M.: Functional Programming with Graphs, *ACM SIGPLAN Int. Conf. on Functional Programming*, 1997. To appear.
ftp://ftp.fernuni-hagen.de/pub/fachb/inf/pri4/papers/fgfl.ps.gz

[7] Erwig, M. & Meyer, B.: Heterogeneous Visual Languages – Integrating Visual and Textual Programming, *IEEE Symp. on Visual Languages*, 1995, 318-325.

[8] Euler, L.: *Briefe an eine deutsche Prinzessin*. Vieweg, 1986.

[9] Harel, D.: On Visual Formalisms, *Communications of the ACM, Vol. 31*, No. 5, 514-530.

[10] Kimura, T.D.: Determinacy of Hierarchical Dataflow Model, Report WUCS-86-5, Washington University, St. Louis, 1986.

[11] Kimura, T.D., Choi, J.W. & Mack, J.M.: Show an Tell: A Visual Programming Language, in E.P. Glinert (ed.): *Visual Programming Environments*, IEEE Computer Science Press, Los Alamitos/CA, 1990, 397-404.

[12] Mosses, P.D.: Denotational Semantics, in J. van Leeuwen (ed.): *Handbook of Theoretical Computer Science, Vol. B*, Elsevier, 1990, 575-631.

[13] Rekers, J. & Schürr, A.: A Graph Grammar Approach to Graphical Parsing, *IEEE Symp. on Visual Languages*, 1995, 195-202.

[14] Rekers, J. & Schürr, A.: A Graph Based Framework for the Implementation of Visual Environments, *IEEE Symp. on Visual Languages*, 1996.

[15] Shin, S.-J.: *The Logical Status of Diagrams*, Cambridge University Press, New York, 1994.

[16] Wang, D. & Lee, J.R.: Visual Reasoning: its Formal Semantics and Applications, *Journal of Visual Languages and Computing, Vol. 4*, No. 4, 1993, 327-356.„