Journal of Visual Languages and Computing (1998) 9, 461–483
Article No. vl980098

Abstract Syntax and Semantics of Visual Languages

MARTIN ERWIG

FernUniversität Hagen, Praktische Informatik IV, 58084 Hagen, Germany
erwig@fernuni-hagen.de

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 little 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. We demonstrate the semantics in action by proving a rule for visual reasoning with Euler circles and by showing the correctness of a Show and Tell program.

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 with 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 [1]. Third, with a precise semantics, various properties of languages can be proved. In particular, we can prove syntactic transformations to be sound with respect to the semantics, for example, 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. In paricular, this applies to heterogeneous or multi-paradigm languages, see, for example, Erwig and Meyer [2].

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 in textual languages: there we can choose among a variety of different semantic formalisms, such as denotational semantics, structured operational semantics, action semantics, evolving

1045-926X/98/050461+23 $30.00/0

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. Abstract Visual Syntax

A textual language L is a set of strings over an alphabet A, that is, $L \subseteq A^*$. 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[a]. 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, that is, 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 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.

A precise definition of abstract syntax does not exist, and it would not make much sense because there are different levels of 'abstractness' that can be dealt with. One reason for abstract syntax not really having found its way into visual languages might be that, as we believe, abstract visual syntax must be 'more abstract' than in the textual case to be helpful. We explain this by a simple example. Consider the following (textual) grammar describing part of a concrete syntax for expressions.

$$
\begin{aligned}
\mathit{expr} &::= \mathit{n\text{-}expr} \mid \mathit{b\text{-}expr} \mid \mathit{if\text{-}expr} \\
\mathit{n\text{-}expr} &::= \mathit{term} \mid \mathit{n\text{-}expr} + \mathit{term} \\
\mathit{term} &::= \mathit{factor} \mid \mathit{term} * \mathit{factor}
\end{aligned}
$$

[a] Relationships with arity > 2 can always be simulated by several binary relationships.

$$\begin{aligned} \textit{factor} &::= \textit{id} \mid (\textit{n-expr}) \\ \textit{b-expr} &::= \textit{id} \mid \textit{b-expr} \vee \textit{b-expr} \mid \ldots \\ \textit{if-expr} &::= \text{if } \textit{b-expr} \text{ then } \textit{expr} \text{ else } \textit{expr} \end{aligned}$$

A corresponding abstract syntax would ignore many details, such as the choice of key words, grammar rules for defining associativity of operators, or rules restricting the typing of operations (see also Mosses [1]):

$$\begin{aligned} \textit{expr} &::= \textit{id} \mid \textit{expr op expr} \mid \text{if } (\textit{expr}, \textit{expr}, \textit{expr}) \\ \textit{op} &::= + \mid * \mid \vee \mid \ldots \end{aligned}$$

This grammar is much more concise. It does not introduce nonterminals for expressions of different types, and it also ignores associativity of operators. (Omitting the key words from the conditional does not make the grammar essentially simpler in this example.) Further operations on sentences of the language can rely on syntax being already checked by a parser and can thus work with the simpler abstract syntax.

In a similar way, the abstract syntax of visual languages need not be concerned with all the details that a concrete syntax specification has to care about; see also Erwig [3]. This means we can abstract from the choice of icons or symbols (comparable to the choice of key words in the textual case) and from geometric details such as size and position of objects (at least up to topological equivalence, that is, as long as relevant relationships between objects are not affected). We can also ignore associativities used to resolve ambiguous situations during parsing much like in the textual example. Moreover, typings of relationships that restrict relationships to specific subsets of symbols can be omitted. This corresponds to grouping operations, such as $+$ or $\vee$, under one nonterminal.

But we can do even more—and this is the point where abstract visual syntax gets more abstract than in the textual case: the above abstract syntax for expressions is still given by a grammar and thus retains some structural information about the language. This is absolutely adequate since the description is very simple and can be easily used when defining, for example, an interpreter for expressions. However, to do so for a visual language requires, in most cases, some effort in the consideration of context information which unnecessarily complicates definitions of transformations. Therefore, we suggest to forget about this structural information, too, and to consider a picture just as a directed, labeled multi-graph where the nodes represent objects and the edges represent relationships between objects. A class of graphs is then just given by two types defining node and edge labels, that is, the types of objects and relationships in the abstractly represented visual language.

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 $\iota : E \to V \times V$ is a total mapping defining for each edge the nodes it connects. The mappings $\nu : V \to \alpha$ and $\varepsilon : E \to \beta$ define the node and edge labels.

V_G and E_G denote the set of nodes and edges of G. The successors of a node are denoted by $succ_G(v)$, which is defined by $succ_G(v) = \{w \in V_G \mid \exists e \in E_G : \iota(e) = (v, w)\}$. Like-

wise, $pred_G(v)$ denotes v's predecessors. Whenever G is clear from the context, we also might simply use V, E, *succ*, and *pred*. We also sometimes use a shorthand for denoting nodes and edges together with their labels: we denote a node (or edge) x with $v(x) = l$ (respectively, $\varepsilon(x) = l$) simply by $x:l$.

The label types α and β might be just 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 $\Gamma(\alpha, \beta)$. In the sequel, we will look at visual languages on this very abstract level, that is, the abstract syntax of a visual language is specified as a set of graphs of a specific type.

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

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 we have demonstrated in [4], 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 further in Section 4. The use of abstract syntax is not restricted to the definition of language semantics, but it can be also used as a basis for transformations between different languages or for mapping between different representations of the same language. This is illustrated in more detail in Erwig [3]. Accidentally, the abstract syntax graphs for the examples used in this paper are all acyclic. This is by no means essential for the presented formalism. Examples for visual languages that have cyclic abstract syntax graphs are state diagrams (syntax and semantics for these are defined in Erwig [3]) or a particular representation of Turing machines (for which syntax and semantics can be found in Erwig [5]).

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.

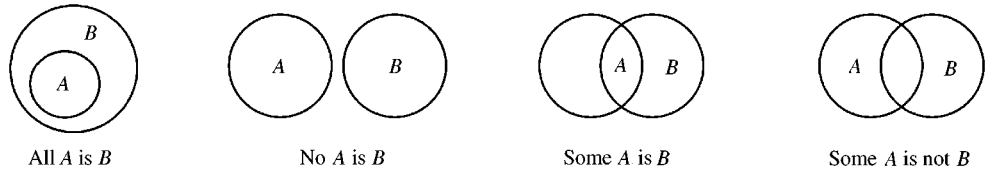


Figure 1. Euler diagrams

3.1. Euler diagrams

The language of Euler diagrams as described by Euler [6] and Shin [7] contains four kinds of basic pictures expressing logical statements as shown in Figure 1. Ambiguities of Euler diagrams and semantic problems arising from these are discussed in detail by Shin [7]. 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 widespread 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 accomplish.

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:
   (a) $A \cap B \neq \emptyset$,
   (b) $A - B \neq \emptyset$, and
   (c) $B - A \neq \emptyset$.
   So this situation describes what we call *proper intersection*, that is, we say A *p-intersects* B.

(2) Is the relative position of labels irrelevant, that is, 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 (that is, 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 with respect to inclusion; we call this relationship *not inclusion-comparable*.

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

Figure 2. Abstract syntax graphs for Euler diagrams

like those shown in Figure 2. 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, that is, nodes of the graph, to subsets of U, that is, $f : V \to 2^U$. Now the semantics can be easily defined:

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

where

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

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:

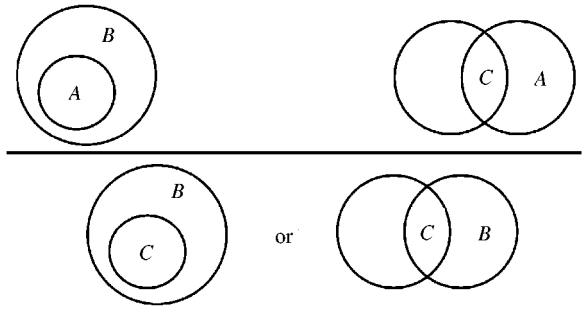
$$\frac{\text{All } A \text{ is } B \qquad \text{Some } C \text{ is } A}{\text{Some } C \text{ is } B}$$

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

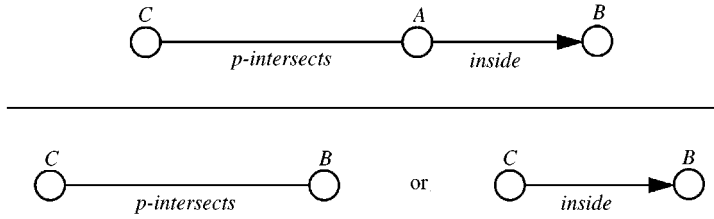
$$\frac{\text{All } A \text{ is } B \qquad \text{Some } C \text{ is } A}{\text{All } C \text{ is } B \text{ or Some } C \text{ is } B}$$

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 logical 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 $C \cap B \neq \emptyset$ (because C is not empty), we know that

$$C \cap B \neq \emptyset \vee C \subseteq B \Leftrightarrow C \cap B \neq \emptyset$$

and second, $C - B \neq \emptyset \vee C \subseteq B$ is always true which can be easily checked by considering all possibilities with respect to the intersection of C and B. Thus, it remains to be shown that

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

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) that $\exists D \neq \emptyset$:

(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. □

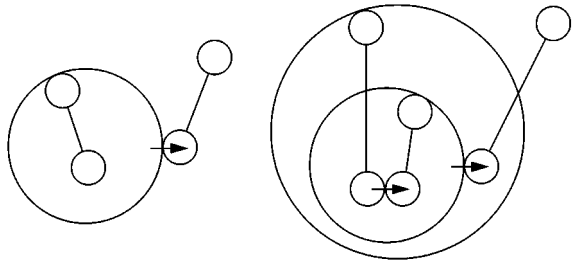


Figure 3. Two VEX expressions

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 [8], 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[b] 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 $\lambda y.((\lambda x.yx)z)$. Now what is the exact meaning of the above drawings? In Citrin *et al.* [8] graphical rewrite rules are given that can be used to reduce VEX pictures to normal forms. This is, however, a purely 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.

[b] Labels are sometimes used for illustration, but strictly, they are not needed.

4.2. Choices of Abstract Syntax

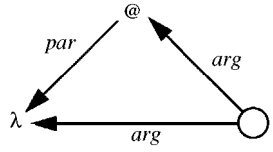
The VEX concrete syntax consists of symbols like circles, lines and arrows, and relationships like inside or touches.

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 (that is, edges labeled with *def*) that lead from variable uses to their definition and by *apply*-edges leading from the expression circles to be applied toward the argument circles. It remains to represent abstractions. An abstraction 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 $(\emptyset, \{def, apply, par, body\})$.

Figure 4 gives the abstract syntax graphs for the VEX pictures from Figure 3. This representation is rather close to the spatial original and should therefore be easy to grasp. However, a DAG representing the lambda-expression in a rather traditional way might be better suited to study, for example, semantics of β-reduction.

Such a representation consists of application-, abstraction- and variable-nodes (with corresponding node labels: @, λ, ○)[c]. 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 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 $(\{@, \lambda\}, \{fun, arg, par, body\})$.

Figure 5 shows the abstract syntax graphs that correspond to the VEX pictures of Figure 3. At this point it is important to recall that the informally stated structural properties are not captured by abstract syntax graphs. This means that the graph 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, that is, syntactically correct pictures.

[c] 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.

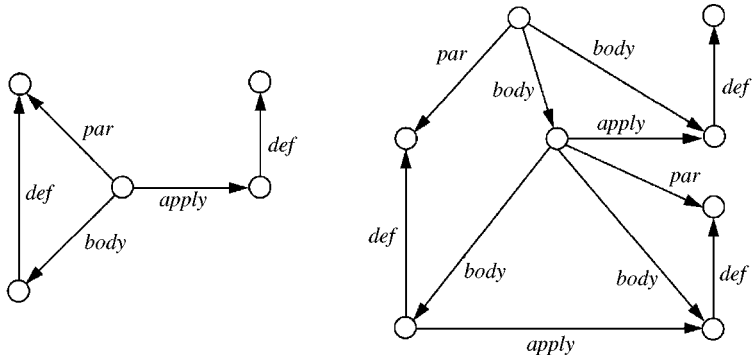


Figure 4. Abstract graphs for VEX expressions

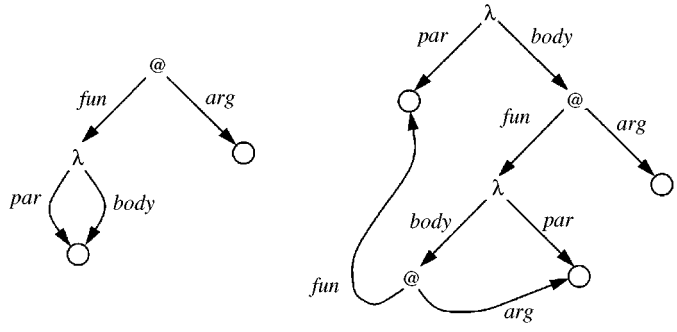


Figure 5. Alternative abstract syntax graphs for VEX

Although the second representation offers advantages in treating certain aspects of semantics, 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 choose 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, for example, 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 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 graph 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 (for example, $d:@$) and *pred-spec* (*succ-spec*) denotes a list[d] of predecessor (successor) nodes possibly extended by labels for the edges that come from (lead to) the nodes. For instance, $[d > fun, e]$ 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.

The first graph from Figure 5 is given by the following expression:

$$N\ ([\ ],\ d:@,\ [\ fun > b,\ arg > c\ ])\ (N\ ([\ ],\ c,\ [\ ])$$

$$(N\ ([\ ],\ b:\lambda,\ [\ par > a,\ body > a])\ (N\ ([\ ],\ a,\ [\ ])\ Empty)))$$

Here a, b, c, and d are arbitrary, pairwise different node identifiers. 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

$$N*\ (d:@,\ [\ fun > b,\ arg > c\ ])\ (c)\ (b:\lambda,\ [\ par > a,\ body > a])\ (a)\ Empty$$

Note that there are, in general, many different graph expressions denoting the same graph, for example, the above term denotes the same graph as

$$N*\ ([d > fun],\ b:\lambda,\ [\ par > a,\ body > a])\ (d:@,\ [arg > c\ ])\ (c)\ (a)\ Empty$$

The relationship between graph expressions and multi-graphs is formally defined as follows:

$$\gamma(Empty) = (\emptyset,\ \emptyset,\ \emptyset,\ \emptyset,\ \emptyset)$$

$$\gamma(N\ ([\ p_1 > x_1, \ldots, p_n > x_n],\ v:l,\ [\ y_1 > s_1, \ldots, y_m > s_m])\ g\ )$$

$$= (V \cup \{v\},\ E \cup \{e_1, \ldots, e_{n+m}\},$$

$$\iota \cup \{(e_1,\ (p_1,\ v)), \ldots, (e_n,\ (p_n,\ v)),\ (e_{n+1},\ (v,\ s_1)), \ldots, (e_{n+m},\ (v,\ s_m))\},$$

$$v \cup \{(v,\ l\ )\},\ \varepsilon \cup \{(e_1,\ x_1), \ldots, (e_n,\ x_n),\ (e_{n+1},\ y_1), \ldots, (e_{n+m},\ y_m)\})$$

where

$$(V,\ E,\ \iota,\ v,\ \varepsilon) = \gamma(g),\ \{e_1, \ldots, e_{n+m}\} \cap E = \emptyset,\ \{p_1, \ldots, p_n, s_1, \ldots, s_m\} \subseteq V, \text{ and } v \notin V$$

Thus, multi-graphs can serve as a kind of normal form for graph expressions. The following result is important, since it guarantees that any graph can be viewed inductively:

[d] 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 simplifies the processing of, for example, successors, in a specific order.

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

We have given the proof in [4]. There we also define 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* [9]: 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 variables. For instance, matching the pattern N $(p, b:l, s)\ g$ against either graph expression from the previous subsection results in the following bindings:

$$p \to [d > \mathit{fun}],\ l \to \lambda,\ s \to [\ \mathit{par} > a,\ \mathit{body} > a\ ],\ g \to \text{`}g\text{-}\mathit{term}\text{'}$$

where *g-term* is an arbitrary representation of the matched graph without node b and its incident edges, for example,

$$\text{`}g\text{-}\mathit{term}\text{'} \cong N * (d:@,\ [\mathit{arg} > c])\ (c)\ (a)\ \mathit{Empty}$$

Formally, graph pattern matching is defined on the basis of the represented multi-graphs. For a given node v assume G can be written as

$$\begin{aligned} G = (&V + \{v\},\ E + \{e_1, \ldots, e_{n+m}\},\\ &\iota + \{(e_1, (p_1, v)), \ldots, (e_n, (p_n, v)), (e_{n+1}, (v, s_1)), \ldots, (e_{n+m}, (v, s_m))\},\\ &\nu + \{(v, l)\},\ \varepsilon + \{(e_1, x_1), \ldots, (e_n, x_n), (e_{n+1}, y_1), \ldots, (e_{n+m}, y_m)\}) \end{aligned}$$

where $S + T$ denotes disjoint set union and where the disjoint union for E is chosen maximally, that is, there is no $e' \in E$ such that there exists $(e', (x, y)) \in \iota$ with $x = v$ or $y = v$. Then matching the pattern $N\ (p, v:l, s)\ g$ to G produces the bindings

$$p \to [\ p_1 > x_1, \ldots, p_n > x_n],\ l \to \mathit{lab},\ s \to [\ y_1 > s_1, \ldots, y_m > s_m],\ g \to (V, E, \iota, \nu, \varepsilon)$$

This means that the meaning of pattern matching does not depend on the representation chosen by a particular graph expression. In other words, we have the freedom to choose graph expressions as we like; we make use of this later on in this paper when we apply semantics definitions to example graphs. Then we shall choose representations that make inductive decompositions of graphs simple so that we need neither transform graph expressions nor map them to the represented multi-graphs.

Patterns can be made more selective 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, for example, we can match the abstraction node b binding the parameter/body node to p/e by using the pattern

$$N\ (b:\lambda,\ [\ \mathit{par} > p,\ \mathit{body} > e\ ])\ g$$

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 ([ ], $b:\lambda$, [$par > p$, $body > e$]) g instead. This, however, fails to match our example graph.

Cascading patterns like $N*\ c_1\ c_2\ \ldots\ c_n\ g$ can be matched against a graph G as follows: let $g_1, \ldots, g_n$ be auxiliary variables to be bound to intermediate decomposed graphs. Now first, $N\ c_1\ g_1$ is matched against G, and the bindings produced by this match, especially the node bindings in c_1 and the rest graph g_1, are then used to match $N*\ c_2 \ldots c_n\ g$ against g_1, that is, $N\ c_2\ g_2$ is matched against g_1, $N\ c_3\ g_3$ is matched against g_2, and so on, until $N\ c_n\ g_n$ is matched against g_{n-1}. Then g is bound to g_n. In this way, $N*$ patterns can actually be used to conveniently find paths (of fixed length) in the 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 (for example, Scott's construction D_∞ or Plotkin's graph model $P\omega$ [10]). 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. It would be also possible to map the abstract syntax to textual lambda-expression and to rely on semantics already defined for the lambda-calculus. However, this would mean one further intermediate representation and, as noted, a sligthly more complicated semantics definition with the need for an environment.

We define the semantics by moving in a controlled way through the abstract graph, that is, semantics are given with respect to 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 (that is, with predecessors r and no successors):

$$S'[\![v,\ N\ (v:\boldsymbol{d})\ g]\!] = \boldsymbol{d}$$

$$S'[\![v,\ N\ (v:@,\ [\ fun > f,\ arg > a\ ])\ g]\!] = S'[\![\ f,\ g]\!]\ (S'[\![a,\ g]\!])$$

$$S'[\![v,\ N*\ (v:\lambda,\ [\ par > p,\ body > b\ ])\ (r,\ p)\ g]\!]$$
$$= \Lambda\boldsymbol{d}.S'[\![b,\ N\ (r,\ p:\boldsymbol{d},\ [\ ])\ g]\!]$$

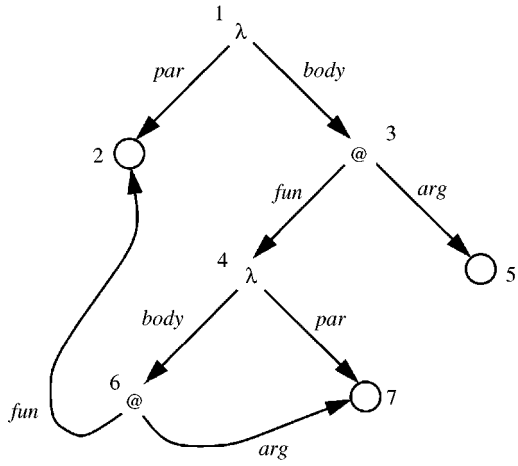


Figure 6. Abstract syntax for lambda expression $\lambda y.((\lambda x.yx)z)$

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

$$root(G) = \{v \in V_G \mid pred_G(v) = [\,]\}$$
$$S[\![G]\!] = S'[\![the(root(G)),\ G]\!]$$

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

We have given an alternative semantics definition for VEX based on the other abstract syntax approach recently [5].

We can use the denotational semantics to 'compute' the meaning for particular VEX expressions. As an example we determine the function denoted by the second VEX picture of Figure 3. For convenience, we repeat the abstract syntax representation with added node identifiers in Figure 6 to facilitate the understanding of the following derivation.

The graph (G_1) is formally defined by the following expressions. The representations are chosen to make subsequent pattern matching easy and to have proper bindings for remaining graphs:

$$G_6 = N^*\ (6:@)\ Empty$$
$$G_4 = N^*\ (4:\lambda,\ [\ par > 7,\ body > 6\,])\ ([6 > arg\,],\ 7)\ G_6$$
$$G_3 = N^*\ (3:@,\ [\ fun > 4,\ arg > 5\,])\ (5)\ G_4$$
$$G_1 = N^*\ (1:\lambda,\ [\ par > 2,\ body > 3\,])\ ([6 > fun\,],\ 2)\ G_3$$

Now the meaning of the graph G_1 is

$$S[\![G_1]\!] = S'[\![the(root(G_1)),\ G_1]\!] = S'[\![1,\ G_1]\!]$$
$$= \Lambda \boldsymbol{d}.S'[\![3,\ N\ ([6 > fun],\ 2:\boldsymbol{d})\ G_3]\!]$$

$$= \Lambda \boldsymbol{d}.(S'[\![4,\ N\ ([6 > \mathit{fun}],\ 2:\boldsymbol{d})\ G_4]\!]\ (S'[\![5,\ N\ (5)\ G_4]\!]))$$

$$= \Lambda \boldsymbol{d}.(\Lambda \boldsymbol{d}'.S'[\![6,\ N^*\ ([6 > \mathit{arg}],\ 7:\boldsymbol{d}'))\ ([6 > \mathit{fun}],\ 2:\boldsymbol{d})\ G_6]\!]\ \bot)$$

$$= \Lambda \boldsymbol{d}.(\Lambda \boldsymbol{d}'.S'[\![6,\ N^*\ (6:@,\ [\mathit{fun} > 2,\ \mathit{arg} > 7])\ (2:\boldsymbol{d})\ (7:\boldsymbol{d}')\ \mathit{Empty}]\!]\ \bot)$$

$$= \Lambda \boldsymbol{d}.(\Lambda \boldsymbol{d}'.(S'[\![2,\ N^*\ (2:\boldsymbol{d})\ (7:\boldsymbol{d}')\ \mathit{Empty}]\!]\ (S'[\![7,\ N^*\ (2:\boldsymbol{d})\ (7:\boldsymbol{d}')\ \mathit{Empty}]\!]))\ \bot)$$

$$= \Lambda \boldsymbol{d}.(\Lambda \boldsymbol{d}'.(\boldsymbol{d}\ \boldsymbol{d}')\ \bot)$$

$$= \Lambda \boldsymbol{d}.\boldsymbol{d}\ \bot$$

Note that $S'[\![5,\ N\ (5)\ G_4]\!] = \bot$ because the semantics of free variables is not defined. Thus, the meaning of the VEX picture is a function that applies its argument to the undefined value.

5. A Larger 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, 12] 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, that is, 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, that is, an inconsistent closed box can be viewed as deleted. With the concept of inconsistency, conditionals can be expressed without having boolean values.

Figure 7 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 (closed) complex box remains consistent, and the '1' can flow directly into the result box. Moreover, the lower (closed) 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.

We choose an abstract syntax that mainly follows the concrete syntax. In particular:

(1) Nodes are labeled by constants (for example, integers), function symbols (such as $+$), $\bigcirc$ (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.)

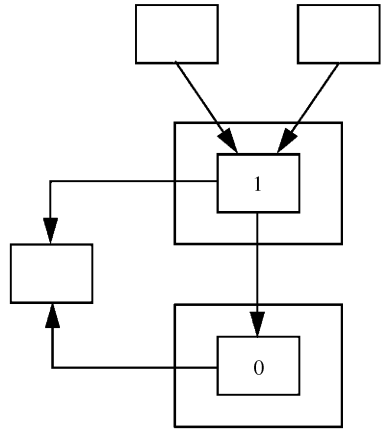


Figure 7. STL program for logical AND

(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. If $j = *$, this means that the complete value flows via the edge.
(3) Each edge $e = (v, w):(i, j)$ (that is, 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 $e_1 = (v, u):(k, j)$ (ending at u) and $e_2 = (x, w):(i, *)$ (connecting x to the target of e).
   (ii) If v is inside u, then $e_1 = (v, x):(1, j)$ and $e_2 = (u, w):(i, k)$.
   Here, k ranges from 1 to n (m) for all n incoming (m outgoing) edges and represents the argument position of the node.
(4) The (top-level) box-graph is extended according to rule (3) as if it were enclosed by a (closed) box having edges ending at the roots and leaving the sinks.

The abstract syntax of the STL program from Figure 7 is shown in Figure 8. For later reference we have added small node numbers to the labels. Nodes with constants as labels are surrounded by circles and can thus be distinguished from newly introduced nodes. Formally, we use integers as labels of newly introduced nodes and quoted integers as constant labels. This means, the label of node 4 is 2 whereas the label of node 8 is $'1$.

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 (let $'\mathrm{IN} = \{'\} \times \mathrm{IN}$):

$$\alpha_0 = (OP \cup \{\bigcirc\} \cup \mathrm{IN} \cup {}'\mathrm{IN}) \times \{open, closed\}$$

$$\beta = \mathrm{IN} \times (\mathrm{IN} \cup \{*\})$$

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:

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

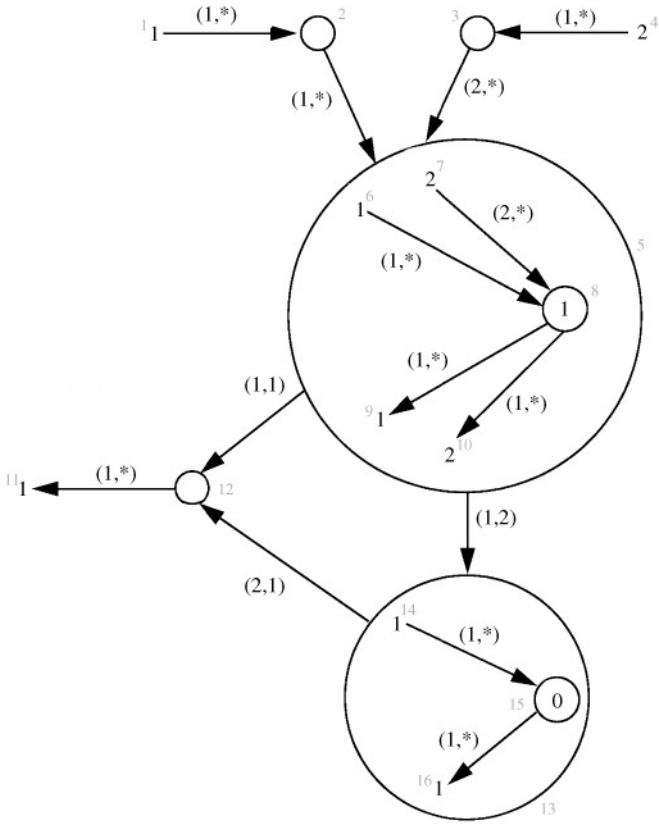


Figure 8. Abstract syntax of the STL program

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

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 $\boldsymbol{D}$ (for example, for integers) and add to it a special value $\diamond$ 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, that is, 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:

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

(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 defined below. In the pattern we assume that the predecessors (p_i) are ordered with respect to 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 $\Pi_*(x) = x$),

$$S''[\![v,\ N\ ([\ p_1 > (1,\ s_1), \ldots,\ p_k > (k,\ s_k)],\ v : f\ )\ g]\!] =$$

$$F[\![\ f\ ]\!](\Pi s_1(S''[\![\ p_1,\ g]\!]), \ldots,\ \Pi s_k(S''[\![\ p_k,\ g]\!]))$$

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 $\diamond$ which is defined to be 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:

$$C'[\![v,\ G]\!] = (open(v) \Rightarrow S''[\![v,\ G]\!] \neq \diamond)$$

$$C[\![G]\!] = \forall v \in V_G\colon C'\ [\![v,\ G]\!]$$

Now the semantics of an STL graph is finally given by

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

If G contains no open boxes, the propagation of inconsistency need not be taken into account because in that case C' and C always yield *true*. Thus, the semantics for graphs without open boxes simplifies to

$$S[\![G]\!] = S'[\![G]\!]$$

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

It remains to define the functions denoted by node labels. An operation 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:

$$F[\![\ f : \boldsymbol{D}^n \to \boldsymbol{D}^m]\!] = f$$

$$F[\![c : \boldsymbol{D}]\!] = \Lambda(\boldsymbol{d}_1, \ldots,\ \boldsymbol{d}_n).\mathit{if}\ \boldsymbol{d}_1 = \cdots = \boldsymbol{d}_n = c\ \mathit{then}\ c\ \mathit{else}\ \diamond$$

$$F[\![\bigcirc]\!] = \Lambda(\boldsymbol{d}_1, \ldots,\ \boldsymbol{d}_n).\mathit{if}\ \boldsymbol{d}_1 = \cdots = \boldsymbol{d}_n\ \mathit{then}\ \boldsymbol{d}_1\ \mathit{else}\ \diamond$$

$$F[\![G : \Gamma]\!] = S[\![G]\!]$$

The first line includes the case for constant labels, that is, $n = 0$. This means in particular, that the definition of S'' reduces in this special case to:

$$S''[\![v,\ N\ ([\ p_1 > (1,\ s_1),\ \ldots,\ p_k > (k,\ s_k)],\ v : \boldsymbol{d})\ g\ ]\!] = \boldsymbol{d}$$

In the reminder of this section, we demonstrate the semantics definition by proving the correctness of the STL program of Figure 7, that is, we want to show that the program indeed computes the logical AND. Let G be any graph expression representing the abstract syntax graph shown in Figure 8. Then we have to prove:

Theorem 2. $S[\![G]\!] = \Lambda(\boldsymbol{d}_1,\ \boldsymbol{d}_2).\mathit{if}\ \boldsymbol{d}_1 = \boldsymbol{d}_2 = 1\ \mathit{then}\ 1\ \mathit{else}\ 0.$

Proof. We use the following abbreviations:

$$G\,|v_1 : l_1, \ldots, v_n : l_n :=$$

$$\mathit{if}\ G = N^*\ (\ p_1,\ v_1,\ s_1) \ldots (\ p_n,\ v_n,\ s_n)\ G'\ \mathit{then}\ N^*\ (\ p_1,\ v_1 : l_1,\ s_1) \ldots (\ p_n,\ v_n : l_n,\ s_n)\ G'\ \mathit{else}\ \perp$$

$$eq := \Lambda(\boldsymbol{d}_1,\ \ldots,\ \boldsymbol{d}_n).\mathit{if}\ \boldsymbol{d}_1 = \cdots = \boldsymbol{d}_n\ \mathit{then}\ \boldsymbol{d}_1\ \mathit{else}\ \diamond$$

$$eq_c := \Lambda(\boldsymbol{d}_1, \ldots,\ \boldsymbol{d}_n).\mathit{if}\ \boldsymbol{d}_1 = \cdots = \boldsymbol{d}_n = c\ \mathit{then}\ c\ \mathit{else}\ \diamond$$

Since G contains no open boxes we can work with the simplified semantics, that is, $S[\![G]\!] = S'[\![G]\!]$. Thus:

$$\begin{aligned}
S[\![G]\!] &= S'[\![G]\!] \\
&= S'[\![N^*\ ([\ ],\ 1:1,\ [(1,*) > 2])\ ([\ ],\ 4:2,\ [(1,*) > 3])\ g]\!] \\
&= \Lambda(\boldsymbol{d}_1,\ \boldsymbol{d}_2).S'[\![G|1:\boldsymbol{d}_1, 4:\boldsymbol{d}_2]\!] \\
&\quad (= \Lambda(\boldsymbol{d}_1,\ \boldsymbol{d}_2).S'[\![N^*\ ([\ ], 1:\boldsymbol{d}_1, [(1,*) > 2]) \\
&\quad\ ([\ ], 4:\boldsymbol{d}_2, [(1,*) > 3])\ g]\!]) \\
&= \Lambda(\boldsymbol{d}_1,\ \boldsymbol{d}_2).S'[\![N\ ([12 > (1,*)],\ 11:1,\ [\ ])\ g_{11}]\!]
\end{aligned}$$

Again we can ignore C and use the simplified definition for S''. Thus, we can continue (omitting brackets around the one-tuple):

$$\begin{aligned}
&= \Lambda(\boldsymbol{d}_1,\ \boldsymbol{d}_2).S''[\![12,\ g_{11}]\!] \\
&= \Lambda(\boldsymbol{d}_1,\ \boldsymbol{d}_2).S''[\![12,\ N\ ([5 > (1,1),\ 13 > (2,1)],\ 12:\bigcirc)\ g_{12}]\!] \\
&= \Lambda(\boldsymbol{d}_1,\ \boldsymbol{d}_2).F[\![\bigcirc]\!]\ (\Pi_1(S''[\![5,\ g_{12}]\!]),\ \Pi_1(S''[\![13,\ g_{12}]\!])) \qquad \text{(A)}
\end{aligned}$$

We next have to determine $S''[\![5,\ g_{12}]\!]$ and $S''[\![13,\ g_{12}]\!]$:

$$\begin{aligned}
S''[\![5,\ g_{12}]\!] &= S''[\![5,\ N\ ([2 > (1,*),\ 3 > (2,\ *)],\ 5:G_5)\ g_5]\!] \\
&= F[\![G_5]\!]\ (\Pi_*(S''[\![2,\ g_5]\!]),\ \Pi_*(S''[\![3,\ g_5]\!])) \\
&= S[\![G_5]\!]\ (S''[\![2,\ g_5]\!],\ S''[\![3,\ g_5]\!]) \qquad \text{(B)}
\end{aligned}$$

To proceed we now need $S''[\![2, g_5]\!]$, $S''[\![3, g_5]\!]$, and $S[\![G_5]\!]$. Note in the following that g_5 and thus all reduced graphs derived from that have their origin in the graph $G|1:\boldsymbol{d}_1,4:\boldsymbol{d}_2$, that is, nodes 1 and 4 have assigned the semantic values (variables) $\boldsymbol{d}_1$ and $\boldsymbol{d}_2$:

$$\begin{aligned} S''[\![2, g_5]\!] &= S''[\![2, N\ ([1 > (1,*)], 2:\bigcirc)\ g_2]\!] \\ &= F[\![\bigcirc]\!]\ (\Pi_*(S''[\![1, g_2]\!])) \\ &= eq\ (S''[\![1, N\ (1:\boldsymbol{d}_1)\ g_1]\!]) \\ &= eq\ (\boldsymbol{d}_1) \\ &= \boldsymbol{d}_1 \end{aligned}$$

The derivation for $S''[\![3, g_5]\!]$ is almost identical and yields

$$S''[\![3, g_5]\!] = \boldsymbol{d}_2$$

For $S[\![G_5]\!]$ we obtain:

$$\begin{aligned} S[\![G_5]\!] = S'[\![G_5]\!] &= S'[\![N * ([\,], 6:1, [(1,*) > 8])\ ([\,], 7:2, [(2,*) > 8])\ g']\!] \\ &= \Lambda(\boldsymbol{d}_3, \boldsymbol{d}_4).S'[\![G_5|6:\boldsymbol{d}_3,7:\boldsymbol{d}_4]\!] \\ &= \Lambda(\boldsymbol{d}_3, \boldsymbol{d}_4).S'[\![N* ([8 > (1,*)], 9:1, [\,])\ ([8 > (1,*)], 10:2, [\,])\ g_9]\!] \\ &= \Lambda(\boldsymbol{d}_3, \boldsymbol{d}_4).(S''[\![8, g_9]\!], S''[\![8, g_9]\!]) \end{aligned}$$

In the next two lines, we give only the values for the first component of the pair, since the second component is identical.

$$\begin{aligned} &= \Lambda(\boldsymbol{d}_3, \boldsymbol{d}_4).(S''[\![8, N\ ([6 > (1,*), 7 > (2,*)], 8:'1)\ g_8]\!], \ldots) \\ &= \Lambda(\boldsymbol{d}_3, \boldsymbol{d}_4).(F[\![8:'1]\!]\ (\Pi_*(S''[\![6, g_8]\!]), \Pi_*(S''[\![7, g_8]\!])), \ldots) \\ &= \Lambda(\boldsymbol{d}_3, \boldsymbol{d}_4).(eq_1(\boldsymbol{d}_3, \boldsymbol{d}_4), eq_1(\boldsymbol{d}_3, \boldsymbol{d}_4)) \end{aligned}$$

We can insert the results for $S''[\![2, g_5]\!]$, $S''[\![3, g_5]\!]$, and $S[\![G_5]\!]$ into (B) and obtain

$$\begin{aligned} S''[\![5, g_{12}]\!] &= \Lambda(\boldsymbol{d}_3, \boldsymbol{d}_4).(eq_1(\boldsymbol{d}_3, \boldsymbol{d}_4), eq_1(\boldsymbol{d}_3, \boldsymbol{d}_4))\ (\boldsymbol{d}_1, \boldsymbol{d}_2) \\ &= (eq_1(\boldsymbol{d}_1, \boldsymbol{d}_2), eq_1(\boldsymbol{d}_1, \boldsymbol{d}_2)) \end{aligned}$$

Next we determine $S''[\![13, g_{12}]\!]$. This works analogous to the derivation of $S''[\![5, g_{12}]\!]$. Since g_{13} is different from g_{12}, we formally have to derive $S''[\![5, g_{13}]\!]$ from anew, but it is obvious that it results in the same function as $S''[\![5, g_{12}]\!]$. So we get

$$\begin{aligned} S''[\![13, g_{12}]\!] &= S''[\![13, N\ ([5 > (1,2)], 13:G_{13})\ g_{13}]\!] \\ &= F[\![G_{13}]\!]\ (\Pi_2(S''[\![5, g_{13}]\!])) \\ &= \Lambda\boldsymbol{d}_5.eq_0(\boldsymbol{d}_5)\ (eq_1(\boldsymbol{d}_1, \boldsymbol{d}_2)) \\ &= eq_0(eq_1(\boldsymbol{d}_1, \boldsymbol{d}_2)) \\ &= \textit{if}\ \boldsymbol{d}_1 = \boldsymbol{d}_2 = 1\ \textit{then}\ \diamond\ \textit{else}\ 0 \end{aligned}$$

To understand the last step consider two cases: if $\boldsymbol{d}_1 = \boldsymbol{d}_2 = 1$, then $eq_1(\boldsymbol{d}_1, \boldsymbol{d}_2) = 1$, and $eq_0(1) = \diamond$. Otherwise, $eq_1(\boldsymbol{d}_1, \boldsymbol{d}_2) = \diamond$, and since $\diamond$ is equal to all values, $eq_0(\diamond) = 0$.

Finally, we can insert $S''[\![5, g_{12}]\!]$ and $S''[\![13, g_{12}]\!]$ into (A) and we obtain (note that Π_1 has no effect on a one-tuple):

$$\begin{aligned} S[\![G]\!] &= \Lambda(\boldsymbol{d}_1, \boldsymbol{d}_2).F[\![\bigcirc]\!]\ (\Pi_1(S''[\![5, g_{12}]\!]),\ \Pi_1(S''[\![13, g_{12}]\!])) \\ &= \Lambda(\boldsymbol{d}_1, \boldsymbol{d}_2).eq\ (eq_1(\boldsymbol{d}_1, \boldsymbol{d}_2),\ \textit{if}\ \boldsymbol{d}_1 = \boldsymbol{d}_2 = 1\ \textit{then}\ \diamond\ \textit{else}\ 0) \\ &= \Lambda(\boldsymbol{d}_1, \boldsymbol{d}_2).\textit{if}\ \boldsymbol{d}_1 = \boldsymbol{d}_2 = 1\ \textit{then}\ 1\ \textit{else}\ 0 \end{aligned}$$

Again, to understand the last step consider the following two cases:

(1) If $\boldsymbol{d}_1 = \boldsymbol{d}_2 = 1$, then $eq_1(\boldsymbol{d}_1, \boldsymbol{d}_2) = 1$ and the second expression yields $\diamond$. Thus the argument pair of eq is $(\diamond, 0)$, and $eq(\diamond, 0) = 1$.

(2) If $\boldsymbol{d}_1 \neq 1$ or $\boldsymbol{d}_2 \neq 1$, then $eq_1(\boldsymbol{d}_1, \boldsymbol{d}_2) = \diamond$, but now the second expression yields 0. Thus the argument pair of eq is $(\diamond, 0)$, and $eq(\diamond, 0) = 0$.

This completes the proof. □

6. Related Work

6.1. Syntax of Visual Languages

There has been quite a lot of work concerning the syntax of visual languages; for an overview, see Marriott *et al.* [13]. However, all these formalisms are concerned with the specification of concrete syntax and address the related aspects of parsing and syntax directed editors.

Only few papers deal with abstract visual syntax. Andries *et al.* [14] and Rekers and Schürr [15, 16] recommend the separation of abstract syntax from concrete syntax. 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 only concerned with translation of visual languages, aspects of semantics definitions are not discussed. More on abstract visual syntax as used in this paper can be found in Erwig [3].

6.2. Semantics of Visual Languages

Besides semantics definitions for specific languages, such as in Kimura [12], there has been not much done about semantics of visual languages in general. Wang and Lee [17] 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.* [18] 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.

The term 'semantics' is sometimes used with a different meaning, for example, in Helm and Marriott [19] it means a set of pictures satisfying a given specification, that is, the semantics is a visual language itself and not a mathematical domain describing the computations performed by a visual language.

6.3. Graph Representation

Using graphs to describe pictures is a common and widespread approach. However, general models that apply to a broad range of visual languages are few. Examples are Harel's higraphs [20] and the theory of graph grammars [21].

Higraphs 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. Higraphs provide a perfect representation for those visual languages that exactly fit that model. However, since higraphs have a fixed structure, their applicability is restricted, and only a certain class of visual languages can be expressed in terms of them. Hence, although quite many applications can, in principle, 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 a fairly general model of visual languages. Graph grammars are very powerful, and they have been extensively used to describe graph transformations. Graph grammars enjoy a large body of theoretical results, and they also provide, in a certain sense, an inductive view of graphs. So why should we need yet another graph model? A major difficulty with graph grammars is that they consider the graphs they operate on as global variables that can be updated destructively. This means that changes performed by grammar rules are implicitly propagated, and thus a declarative treatment of graphs is prohibited. Things are complicated by the fact that the semantics of graph grammars themselves is rather complex due to advanced embedding rules and nondeterminism. In contrast, the inductive graph view presented in this paper is quite simple, and it treats graphs as explicit parameters of transformations.

7. Conclusions and Future Work

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.

A drawback of the approach presented so far is that visual information is mapped completely to a textual description. We are currently extending the formalism by a heterogeneous, that is, semi-visual, notation so that certain relationships, such as adjacency or intersection, need not be encoded in graph edges, but can be kept in visual form [5]. This will make semantics definitions and other transformations much more readable.

References

1. P. D. Mosses (1990) Denotational semantics. In: *Handbook of Theoretical Computer Science, Vol. B* (J. van Leeuwen, ed.). Elsevier, Amsterdam, pp. 575-631.

2. M. Erwig & B. Meyer (1995) Heterogeneous visual languages—integrating visual and textual programming. *IEEE Symposium on Visual Languages*, Darmstadt, Germany, pp. 318–325.
3. M. Erwig (1997) Abstract visual syntax. *Proceedings of the 2nd IEEE International Workshop on Theory of Visual Languages*, Capri, Italy, pp. 15–25.
4. M. Erwig (1997) Functional programming with graphs. *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming*, Amsterdam, The Netherlands, pp. 52–65.
5. M. Erwig (1998) Visual semantics—or: what you see is what you compute. *IEEE Symposium on Visual Languages*, Halifax, Nova Scotia (to appear).
6. L. Euler (1986) *Briefe an eine deutsche Prinzessin*. Vieweg, Germany.
7. S.-J. Shin (1994) *The Logical Status of Diagrams*. Cambridge University Press, New York, 197pp.
8. W. Citrin, R. Hall & B. Zorn (1995) Programming with Visual Expressions. *IEEE Symposium on Visual Languages*, Darmstadt, Germany, pp. 294–301.
9. M. Erwig (1996) Active patterns. *Proceedings of the 8th International Workshop on Implementation of Functional Languages*, Bonn, Germany, Lecture Notes in Computer Science, Vol. 1268. Springer, Berlin, pp. 21–40.
10. H. P. Barendregt (1981) *The Lambda Calculus—Its Syntax and Semantics*. North-Holland, Amsterdam, 615pp.
11. T. D. Kimura, J. W. Choi & J. M. Mack (1990) Show and tell: a visual programming language. In: *Visual Programming Environments* (E. P. Glinert, ed.) IEEE Computer Science Press, Los Alamitos/CA, pp. 397–404.
12. T. D. Kimura (1986) Determinacy of hierarchical dataflow model. Report WUCS-86-5, Washington University, St. Louis.
13. K. Marriott, B. Meyer & K. Wittenburg (1996) A survey of visual language specification and recognition. *Workshop on Theory of Visual Languages*, Boulder, CO.
14. M. Andries, G. Engels & J. Rekers (1996) How to represent a visual program? *Workshop on Theory of Visual Languages*, Boulder, CO.
15. J. Rekers & A. Schürr (1995) A graph grammar approach to graphical parsing. *IEEE Symposium on Visual Languages*. Darmstadt, Germany, pp. 195–202.
16. J. Rekers & A. Schürr (1996) A graph based framework for the implementation of visual environments. *IEEE Symposium on Visual Languages*, Boulder, CO.
17. D. Wang & J. R. Lee (1993) Visual reasoning: its formal semantics and applications. *Journal of Visual Languages and Computing* 4, 327–356.
18. P. Bottoni, M. F. Costabile, S. Levialdi & P. Mussio (1995) Formalising visual languages. *IEEE Symposium on Visual Languages*, Darmstadt, Germany, pp. 45–52.
19. R. Helm & K. Marriott (1991) A declarative specification and semantics for visual languages. *Journal of Visual Languages and Computing* 2, 311–331.
20. D. Harel (1988) On visual formalisms. *Communications of the ACM* 31, 514–530.
21. B. Courcelle (1990) Graph rewriting: an algebraic and logic approach. In: *Handbook of Theoretical Computer Science, Vol. B* (J. van Leeuwen, ed.). Elsevier, Amsterdam, pp. 193–242.