Visual Graphs

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

Abstract

The formal treatment of visual languages is often based on graph representations. Since the matter of discourse is visual languages, it would be convenient if the formal manipulations could be performed in a visual way. We introduce visual graphs to support this goal. In a visual graph some nodes are shown as geometric figures, and some edges are represented by geometric relationships between these figures. We investigate mappings between visual and abstract graphs and show their application in semantics definitions for visual languages and in formal manipulations of visual programs.

1. Introduction

Graphs are widely used as a formal representation of visual languages with nodes denoting objects and edges depicting relationships between objects. In a sense, graphs are a natural choice because the generalization of textual languages to visual languages essentially means to move from symbols related only by one relationship (the linear ordering given by sequencing) to multiple relationships which are perfectly represented in a graph by labeled edges between nodes representing symbols.

Now when an edge represents a simple relationship, such as *inside*, this edge can, in some cases, be displayed by showing the connected nodes in the represented relationship. This leads to representations that are more closely related to the original picture, and this can greatly simplify the understanding and the formal manipulation of such graphs. We formalize this idea by defining *visual graphs* which have a subset of nodes and edges that are depicted in a visual way. We define mappings from visual graphs to their corresponding abstract graphs, which is not a trivial task since visual graphs generally show more relationships than are actually given in the abstract graph. We investigate conditions that ensure the existence of unambiguous such mappings. Once a mapping has been established, we can use visual graphs in formal graph manipulations. This results in transformations that are much clearer and much easier to understand while still based, by virtue of that mapping, on the abstract representation.

Thus, visual graphs are, in essence, a visual language for graphs. The importance lies in its possible large scope: whenever we have to deal (formally) with a visual language – and this is likely to be done on its formal graph representation – we can use visual graphs as a visual language to reason about visual languages at a high level.

In the following we use as a running example VEX, a visual notation for the lambda calculus [4], and consider as formal manipulations semantics definitions and the computation with VEX pictures. We chose VEX because on the one hand, it is a powerful (i.e., computationally complete) and therefore interesting language, and on the other hand, it is manageable because it is relatively small, i.e., it consists of only few visual concepts. A further interesting application area for visual graphs is to give visually enriched semantics definitions for traditionally textual formalisms. This has been demonstrated for Turing Machines in [7].

The paper is structured as follows: in the next section we describe several aspects of the use of graphs to abstractly represent visual languages. Section 3 presents visual graphs as a semi-abstract notation, and in Section 4 we show how to map between semi-abstract and abstract representations. Section 5 demonstrates how visual graphs can be used in semantics definitions, and Section 6 shows how the framework can help with formal manipulations of picture graphs. Related work is discussed in Section 7, and conclusions follow in Section 8.

2. Abstract graph representations

Assuming that a picture is represented by a graph, a visual language is simply a set of such graphs. We will illustrate the following definitions with VEX (more details and explanations can be found in [8]). In VEX, empty circles represent identifiers, non-empty circles denote abstractions, and an application is depicted by two externally tangent circles with an arrow at the tangent point leading from the function to the argument. Each empty circle representing the use of an identifier is connected by a straight line to a so-called root node, which is either internally tangent to an abstraction circle and represents then the parameter of the abstraction or which lies outside any other circle and then

represents a free variable. Figure 1 shows on the left a VEX expression for the lambda term $\lambda y.((\lambda x.yx)z)$.

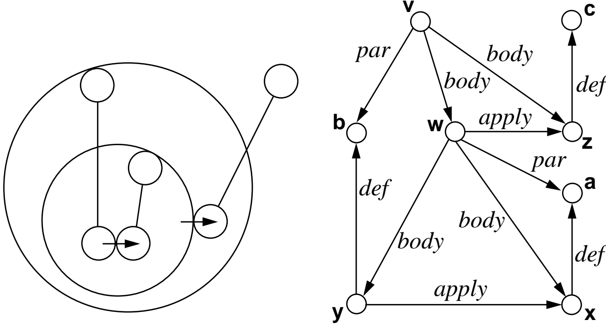


Figure 1: A VEX expression and its abstract syntax graph

2.1. Abstract graph syntax for visual languages

Depending on the intended use, visual language representations are free to ignore more or less details of the concrete appearance. As an example, we consider how to define semantics of visual programming languages. Hence, we can well work with *abstract visual syntax* which allows us to abstract from many details and problems that arise in parsing visual languages. In particular, we can forget about all structural constraints and simply regard a visual language as a set of graphs. 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 $succ_G(v) = \{w \in V_G \mid \exists e \in E_G\colon \iota(e) = (v, w)\}$. Likewise, $pred_G(v)$ denotes v's predecessors. The set of all graphs of type (α, β) is denoted by $\Gamma(\alpha, \beta)$, and a *visual language of type* (α, β) is defined as a set of graphs $VL \subseteq \Gamma(\alpha, \beta)$.

Let us define an abstract syntax for VEX. We replace each line connecting an identifier node v with its definition w by a *def*-edge (that is, an edge labeled with *def*) from v to w, and an external adjacency relationship together with an arrow is replaced by an *apply*-edge leading from the expression circle to be applied toward the argument circle. 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 $e_1, \ldots, e_n$ (non-transitively) 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 $(c, e_1), \ldots, (c, e_n)$. Node labels are not needed, and thus the abstract syntax for VEX is given by graphs of type (*Unit*, {*def*, *apply*, *par*, *body*}) where *Unit* = {()} (we omit unit labels "()" in pictures). Figure 1 shows on the right the abstract syntax graph for the VEX picture on the left.

As in the textual case the choice of abstract syntax for a visual language is by no means unique. An alternative abstract syntax for VEX can be found in [8].

2.2. Inductive graphs and pattern matching

In order to define, for instance, denotational semantics for a visual language we need a structured way of accessing all the elements of an abstract syntax graph. This is offered by an inductive view of graphs that allows the structured, step-by-step graph decomposition: 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. In this way graphs can be built by the constant *Empty* and expressions *N node-context g* where *node-context* is a triple (*pred-spec*, *node-spec*, *succ-spec*). Here, *node-spec* is a node identifier not already contained in g followed by a label (for example, a:()) and *pred-spec* (*succ-spec*) denotes a list of predecessor (successor) nodes extended by labels for the edges that come from (lead to) the nodes. For instance, [*d*›*apply*] denotes a list of a predecessor node d where the edge coming from d has label *apply*. Similarly, [*par*›*b*, *body*›*b*] denotes a single successor b that is reached via two differently labeled edges. We denote concatenation of a single element or list x with another list l by $[x \mid l]$. In the sequel we may omit empty sequences and unit labels, and we abbreviate a cascade of N-constructors by a single N^*-constructor. If $l = [v_1, \ldots, v_n]$, we also abbreviate [*lab*›v_1, …, *lab*›v_n] by [*lab*»*l*] (the same for predecessors). For instance, the graph from Figure 1 is given by the expression:

*N** (*v*, [*par*›*b*, *body*»[*w*, *z*]])
(*w*, [*par*›*a*, *body*»[*y*, *x*], *apply*›*z*]) (*z*, [*def*›*c*]) (*c*)
(*y*, [*def*›*b*, *apply*›*x*]) (*b*) (*x*, [*def*›*a*]) (*a*) *Empty*

We also use an additional edge constructor *E* (*v*›*l*›*w*) *g* which simply inserts an edge with label l between the two nodes v and w (which must be already present in g).

It is not difficult to prove that each directed labeled multi-graph can be represented by a graph expression [6]. Moreover, there are, in general, many different graph expressions denoting the same graph. In particular, for each node v contained in g there is always an expression for g that inserts v last. This is important since it makes a powerful kind of pattern matching [5] applicable to graphs: a pattern is a graph expression containing variables. Matching a pattern like *N* (*p*, *v*:*l*, *s*) *g* to a graph expression binds the components of 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, if v is bound to y, then matching the pattern $N\ (p,\ v{:}l,\ s)\ g$ against the above graph expression results in the bindings: $p \rightarrow [w \rangle body]$, $l \rightarrow ()$, $s \rightarrow [def \rangle b,\ apply \rangle x]$, and $g \rightarrow$ *rest-graph* where *rest-graph* is a term/expression denoting g without y and its incident edges.

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. Moreover, a sub-pattern *lab*»*s* (*p*»*lab*) binds to *s* (*p*) the list of all successors (predecessors) that are connected via *lab*-edges to the current node. An edge pattern matches the mentioned edge and the incident nodes, but removes only the edge. In function definitions consisting of several cases patterns are tried in the order of occurrence. Thus, pattern matching is deterministic (except for the order of nodes in generated list bindings). Some examples follow later.

2.3. Denotational semantics

With a suitable domain $\boldsymbol{D}$ for the lambda-calculus we can define the semantics of VEX by inductively processing abstract syntax graphs. We define a semantic function S' that takes a graph and a node specifying the decomposition order. We have three cases to consider: first, the semantics of an identifier that is connected by a *def*-edge to a parameter node carrying a semantic value is just that value. Second, the meaning of a node v connected by an *apply*-edge to node w is given by applying the semantics of v, which is expected to be a function value, to the value denoted by w. Finally, the semantics of an abstraction is a function value (Λ denotes the semantic abstraction function) which maps any value $\boldsymbol{d}$ from $\boldsymbol{D}$ to the value denoted by the body of the abstraction when the parameter node is labeled $\boldsymbol{d}$. If the body contains two or more nodes, these have to be chained by *apply*-edges, and we have to take the semantics of the leftmost node since application associates to the left. (Remarks: (1) In VEX, the evaluation order can be changed by writing priority values above the application arrows. We ignore this here for brevity. (2) The use of the edge pattern in the second line preserves v from being removed from the matched graph. (3) In order to change the label of the parameter node p to $\boldsymbol{d}$ we have to decompose p from the graph and reinsert it with the new label and the old context, that is, with predecessors r and no successors.)

$$S'[\![v,\ N^*\ (v,\ [def \rangle w])\ (w{:}\boldsymbol{d})\ g]\!] = \boldsymbol{d}$$
$$S'[\![v,\ E\ (v \rangle apply \rangle w)\ g]\!] = S'[\![v,\ g]\!]\ (S'[\![w,\ g]\!])$$
$$S'[\![v,\ N^*\ (v,\ [par \rangle p \mid body \gg b])\ (r,\ p)\ g]\!] = \Lambda \boldsymbol{d}.S'[\![leftmost(b,\ g),\ N\ (r,\ p{:}\boldsymbol{d},\ [\,])\ g]\!]$$

The function *leftmost* traverses the list b until a node v is found that does not have an incoming *apply*-edge, which means, that v does not match the first pattern:

$$leftmost([\ v \mid l],\ E\ (u \rangle apply \rangle v)\ g) = leftmost(l,\ g)$$
$$leftmost([\ v \mid l],\ g) = v$$

Now the semantics of a VEX-graph g is finally given by:

$$S[\![g]\!] = S'[\![root(g),\ g]\!]$$

where $root(g) = v \in V_g : pred_g(v) = [\,]$. For a different VEX semantics based on an alternative abstract syntax, see [8].

3. Visual graphs and semi-abstract syntax

The textual semantics definition has one major disadvantage: the notation is not directly related to the original language, and to understand the meaning of a picture one has to perform two non-trivial (mental) mappings: (1) translate pictures to abstract syntax and (2) map graphs to semantic values. This situation can be improved if the abstract syntax graphs, which are the “glue” for the two mappings, can be lifted onto a more visual level. This is where visual graphs come into play.

In a first step we can immediately give a visual versions of semantics by using a straightforward visualization for graphs. However, this does not really mean a big advance. We can obtain much better results if we display nodes by symbols used in the original visual language and if edges representing concrete geometric relationships are replaced in the picture by displaying the participating nodes (that is, their representing symbols) such that exactly these relationships do hold. So, in a sense, we go back from abstract syntax to concrete syntax, but only partially: some abstract relationships or relationships having very complex visualization rules can (and should) still be depicted as simple graph edges. Hence we are working with a special kind of graphs which we call *visual graphs*. Strictly speaking, a visual graph is just a graph as defined in Section 2.1 where a subset of node and edge labels, namely the geometric ones, have a special meaning, that is, they have their own specific visualizations. The use of visual graphs has two aspects: first, layout algorithms can be used to visualize syntax graphs. We shall not consider this here. Second, visual representations of graphs can be used in formal manipulations, and as (textual) graphs are a model for abstract visual syntax, visual graphs provide a concept of *semi-abstract visual syntax*.

In the case of VEX we keep the original *inside* and *adjacent* relationships in favor of the more abstract *par*-, *body*-, and *apply*-edges, but we still use *def*-edges to represent identifier definition/use relationships more directly. On the other hand, we drop arrows and instead use a *left-of* relationship to represent an ordering among circles participating in a nested application. Thus, we use graphs of type

({*circle*}, {*def*, *adjacent*, *inside*, *left-of*}) as a semi-abstract VEX-syntax. (The decision whether a relationship should be visualized or stay abstract is guided by the corresponding visualization and representation mappings, see next section.) We assume that transitive relationships are represented in graphs in transitively reduced fashion, that is, a transitive relationship R between nodes v and w is *not* represented by an edge if there is a further node u, such that R holds between v and u and also between u and w.

Figure 2 shows the semi-abstract syntax graph and its visualization for the VEX picture from Figure 1. (We display two directed edges (v, w) and (w, v) having the same label as one undirected edge, and for readability we omit all *left-of* edges between non-adjacent objects, that is, after transitive reduction we omit the *left-of* edges (**b**, **a**), (**b**, **x**), (**y**, **a**), (**a**, **z**), (**x**, **z**), and (**z**, **c**).)

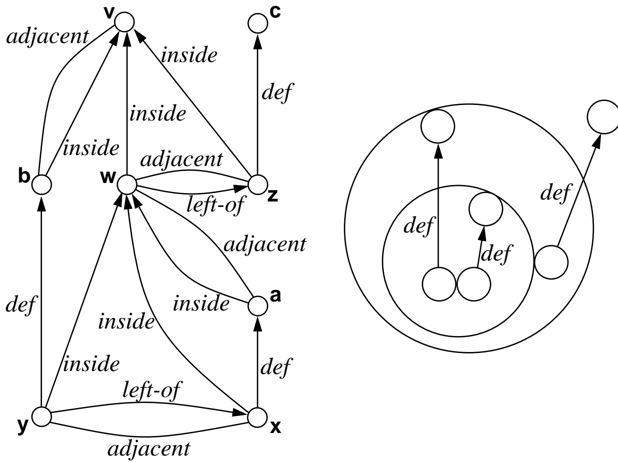
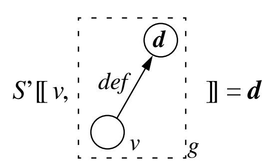


Figure 2: Semi-abstract VEX syntax graph

Now it seems that we can immediately give a visual semantics for VEX using visual instead of textual graphs. However, there remains an important problem: semi-abstract notation displays, in general, more relationships than needed or wanted. In Figure 2, for instance, the upper parameter node **b** is certainly *left-of* the lower parameter node **a**, but this relationship is not important for the semantics although *left-of* is a relevant relationship with regard to nodes being part of an application. In other words, there are relationships in a picture that are relevant only in specific contexts. The problem is that if we use a pattern containing such implicit relationships in an equation, we overly restrict the applicability of that equation, and this over-specification might cause undesired partiality in the definition. For example, if we try to use a visual graph in the first semantic equation for VEX (the details of the notation are not important here and will be explained later), then this equation defines semantics for only those variables whose use appears *left-of* the definition. Although this could principally be remedied by supplying additional equations cover-



ing all possible cases,[1] this approach unnecessarily complicates formal definitions and is prone to errors. Moreover, the number of needed equations can grow significantly.

A solution is to formally keep definitions on the basis of abstract graphs, but to actually use visual graphs in equations. This becomes possible by defining appropriate mappings between abstract and visual graphs. When properly defined, these mappings allow to use semi-abstract syntax within equations while interpreting certain geometric relationships only in specific contexts.

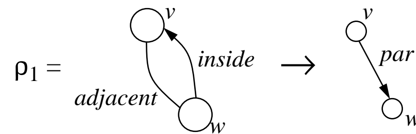
4. Graph visualization and representation

Mappings between abstract and visual graphs can be conveniently expressed by graph rewrite systems. (Since typically several relationships are mapped to one abstract graph edge, such mappings cannot be directly given by graph homomorphisms.) In the following we always denote by $\Gamma(\alpha', \beta')$ an abstract graph type and by $\Gamma(\alpha, \beta)$ the corresponding visual graph type. Thus, in the case of VEX we have $\alpha' = Unit$, $\alpha = \{circle\}$, $\beta' = \{def, apply, par, body\}$, and $\beta = \{def, adjacent, inside, left\text{-}of\}$.

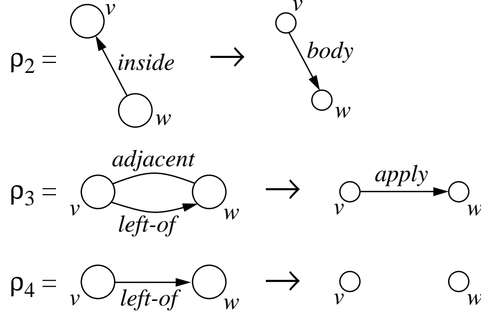
4.1. Typed graph rewriting

The idea is to define the visualization of abstract graphs $\phi^* : \Gamma(\alpha', \beta') \rightarrow \Gamma(\alpha, \beta)$ and, in particular, the representation of visual graphs $\rho^* : \Gamma(\alpha, \beta) \rightarrow \Gamma(\alpha', \beta')$ in several steps: first, we define rule systems $\rho : \Gamma(\alpha, \beta) \rightarrow \Gamma(\alpha', \beta')$ and $\phi : \Gamma(\alpha', \beta') \rightarrow \Gamma(\alpha, \beta)$ for transforming subgraphs, then we define induced rewrite relationships, and we finally restrict to rewrite relationships that are functions.

We only need a restricted form of graph rewriting where a single rewrite rule is given by a pair of graphs (L, R) that have the same node set. Then, informally, a rewrite step finds an occurrence of L in a graph G, removes the edges of L in G, adds the edges of R to G, and relabels the nodes in G by the node labels of R. For example, the representation of visual VEX graphs by abstract syntax graphs is defined by the following rule system $\rho_{VEX} = \{\rho_1, \rho_2, \rho_3, \rho_4\}$:



1. We could add two more equation displaying the node v directly under, respectively, to the right of the node containing ***d***. We could also simply use one picture in which the *left-of* relationship does not hold. But having to be aware of relationships that must be avoided complicates the definition process considerably.



The last rule is for clearing up: *left-of* relationships between nodes that are not adjacent have to be simply removed from the graph. The above notation is already an abbreviation. Formally, we have to define, say ρ_1, as:[1]

$((V_L, E_L, \iota_L, \nu_L, \varepsilon_L), (V_R, E_R, \iota_R, \nu_R, \varepsilon_R))$ with
$V_L = V_R = \{v, w\}$, $E_L = \{e_1, e_2, e_3\}$, $E_R = \{e_4\}$
$\iota_L(e_1) = \iota_R(e_4) = (v, w)$, $\iota_L(e_2) = \iota_L(e_3) = (w, v)$
$\nu_L(v) = \nu_L(w) = circle$, $\nu_R(v) = \nu_R(w) = ()$
$\varepsilon_L(e_1) = \varepsilon_L(e_2) = adjacent$, $\varepsilon_L(e_3) = inside$,
$\varepsilon_R(e_4) = par$

Next we will define how a graph rewrite systems can be applied. Let $r = \{r_1, \ldots, r_k\}$ be a rewrite system and $r_i = (L, R) \in r$ an arbitrary single rewrite rule. An *r_i-match* for a graph G is a mapping $\mu : L \rightarrow G$ such that:

(1) $\forall\, v \in V_L$: $\nu_G(\mu(v)) = \nu_L(v) \vee \nu_G(\mu(v)) \in rng(\nu_R)$
(2) $\forall\, e \in E_L$: $\varepsilon_G(\mu(e)) = \varepsilon_L(e) \wedge \iota_G(\mu(e)) = \mu(\iota_L(e))$
(3) μ is injective

Note: the second condition of (1) allows nodes that are already relabeled to be matched again. This enables the rewriting of edges incident to one node in different steps by different rules. In (2) we use the convention that $\mu(v, w) = (\mu(v), \mu(w))$. Thus, the second condition requires that if $\mu(e) = e'$ and if $\iota_L(e) = (v, w)$ and $\iota_G(e') = (v', w')$, then it must hold that $\mu(v) = v'$ and $\mu(w) = w'$.

Given an r_i-match μ, we say G *rewrites* by r_i in one step into $G' = (V', E', \iota', \nu', \varepsilon')$, denoted by $G \Rightarrow_{r_i} G'$ (or just $G \Rightarrow_r G'$ if the rule applied is not of interest), where:

(*a*) $V' = V_G$ and $E' = E_G - \mu(E_L) \cup E_R$
(*b*) $\iota' = \iota_G \setminus \mu(E_L) \cup \mu \circ \iota_R$
(*c*) $\nu' = \nu_G \setminus \mu(V_L) \cup \nu_R \circ \mu^{-1}$ and $\varepsilon' = \varepsilon_G \setminus \mu(E_L) \cup \varepsilon_R$

Remarks:

(*a*) Nodes are not changed, those edges in G are removed that match edges in L, and edges from R are added (note that $\mu(X) = \{\mu(x) \mid x \in X\}$).

(*b*) Undefine incidences for deleted edges, and add incidences for R-edges. Note that $f \setminus X = \{(x, y) \in f \mid x \notin X\}$. Thus, since $\mu(E_L)$ denotes the set of edges from G matching L, $\iota_G \setminus \mu(E_L)$ restricts G's incidence mapping to those edges that have not been matched. Moreover, we have $g \circ f = \{(x, g(y)) \mid (x, y) \in f\}$. This means that $\mu \circ \iota_R$ adds incidences for the edges of R using, however, the node identifiers of G.

(*c*) Undefine labels for L-nodes and L-edges, and redefine labels as given by R. ν_R must be composed with μ^{-1} since it is defined on V_R, whereas ν' must be defined on $V' = V_G$. On the other hand, E_R is directly added to E_G, so ε_R need not be adjusted.

Now we say that G *rewrites to* G', denoted by $G \Rightarrow_r^* G'$, if there are graphs $G_0, \ldots, G_n$ ($n > 0$), such that $G = G_0 \Rightarrow_r G_0 \Rightarrow_r \ldots \Rightarrow_r G_n = G'$.

Note that the rewrite relationships $\Rightarrow_\rho$ and $\Rightarrow_\phi$ are defined on $\Gamma(\alpha \cup \alpha', \beta \cup \beta')$. The same holds for the respective closures $\Rightarrow_\rho^*$ and $\Rightarrow_\phi^*$. Since we are interested in the extreme values of rewrite chains, that is, (α, β)-graphs and (α', β')-graphs, we also define: (i) G *is represented by* G', denoted by $G \Rightarrow_\rho^\bullet G'$, if $G \Rightarrow_\rho^* G'$, $G \in \Gamma(\alpha, \beta)$, and $G' \in \Gamma(\alpha', \beta')$, and (ii) G' *is visualized by* G, denoted by $G' \Rightarrow_\phi^\bullet G$, if $G' \Rightarrow_\phi^* G$, $G' \in \Gamma(\alpha', \beta')$, and $G \in \Gamma(\alpha, \beta)$.

Let us consider as an example a ρ_1-match for the graph from Figure 2. There we have omitted edge identifiers, so let us assume we have $\{e_a, e_b, e_c\} \subset E_G$ with $\iota_G(e_a) = (\mathbf{v}, \mathbf{b})$, $\iota_G(e_b) = \iota_G(e_c) = (\mathbf{b}, \mathbf{v})$, and $\varepsilon_G(e_a) = \varepsilon_G(e_b) = adjacent$ and $\varepsilon_G(e_c) = inside$. Actually, there are two possibilities to give a match. For the outer abstraction, we get for μ: $\mu(v) = \mathbf{v}$, $\mu(w) = \mathbf{b}$, $\mu(e_1) = e_a$, $\mu(e_2) = e_b$, $\mu(e_3) = e_c$. It is easy to see that μ satisfies the conditions (1) to (3). Now G rewrites by ρ_1 in one step to $G' = (V', E', \iota', \nu', \varepsilon')$ with:

$V' = V_G$, $E' = E_G - \{e_a, e_b, e_c\} \cup \{e_4\}$
$\iota' = \iota_G \setminus \{e_a, e_b, e_c\} \cup \{(e_4, (\mathbf{v}, \mathbf{b}))\}$
$\nu' = \nu_G \setminus \{\mathbf{v}, \mathbf{b}\} \cup \{(\mathbf{v}, ()), (\mathbf{b}, ())\}$
$\varepsilon' = \varepsilon_G \setminus \{e_a, e_b, e_c\} \cup \{(e_4, par)\}$

Again it is easy to check that this graph results from G by replacing the *inside* and *adjacent* edges between **v** and **b** by a *par*-edge and that it conforms to the conditions (*a*) to (*c*). We can show that G rewrites to the abstract graph from Figure 1 by giving a concrete sequence of rewrite rules which are identified simply by their rule numbers (by i^k we mean rule i applied k times): $1^2 2^4 3 4^6$. Of course, this is not the only possible sequence, however, the order of applying rules is not completely free. We address this issue next.

4.2. Properties of typed graph rewriting

Of particular interest are representation relationships $\Rightarrow_\rho^\bullet$ which are right-univalent (or, functional) since they map visual graphs uniquely to abstract syntax graphs. They

---

1. Note that bootstrapping the formalism yields visual rewrite systems, and we can specify, for example, ρ_1, by: (w inside v) $\rightarrow$ v —par→ w

can thus be used to lift graph equations onto the visual level. If $\Rightarrow_\rho^\bullet$ is functional, we denote the function also by ρ^*, and we also say that the rewrite system ρ is *eventually confluent*, or just, *e-confluent*.

We can then use visual graphs, for example, in semantics definitions by writing equations, such as

$$S[\![\, \rho^*(G)\,]\!] = e$$

which means that the visual graph G is immediately mapped to its abstract equivalent G', and S actually works on the abstract graph G'. The rewrite system ρ allows us to use pictures (that is, visual graphs) in the definition, and the fact that ρ is e-confluent guarantees that this use of visualization does not introduce impreciseness by partiality, ambiguity, etc.

Now why does $S[\![\, \rho^*(G)\,]\!]$ work whereas $S[\![\, G\,]\!]$ fails in general? It is because ρ^* is defined to leave certain irrelevant relationships (which are implicit in the visualization) unconsidered. In the VEX semantics, for example, this is the case for the *left-of* relationships of non-adjacent circles.

E-confluence is a weaker notion than confluence since it requires confluence only on "final" values. In any case, before we can use visual graphs, we have to be sure that the associated representation relationship is functional. The restricted form of rewrite systems considered has some properties that can help to simplify e-confluence proofs: most importantly, since no nodes can be deleted or generated, we can simply focus on the replacement of edges.

We define the *domain* and *range* of a rewrite rule as the set of node and edge labels of its left/right graph component, that is, for a rewrite rule $r_i = (L, R)$ we let $dom(r_i) = (dom(\nu_L), dom(\varepsilon_L))$ and $rng(r_i) = (rng(\nu_R), rng(\varepsilon_R))$, and the *domain*/*range* of a rewrite system is defined as the union of domains/ranges of all rules, that is:

$$dom(r) = (\cup_{r_i \in r}\, \pi_1(dom(r)), \cup_{r_i \in r}\, \pi_2(dom(r)))$$
$$rng(r) = (\cup_{r_i \in r}\, \pi_1(rng(r)), \cup_{r_i \in r}\, \pi_2(rng(r)))$$

Here π_i selects the ith component from a tuple.

Next we define a notion of monotonicity. A rewrite system r is *monotone* if $\pi_1(dom(r)) \cap \pi_1(rng(r)) = \varnothing$ and if $\pi_2(dom(r)) \cap \pi_2(rng(r)) = \varnothing$. This means, edges that are added by a monotone rewrite system cannot be removed, and nodes are relabeled at most once. A direct consequence of this is:

Theorem 1. *Monotone rewrite systems are terminating.*

For instance, ρ_{VEX} is monotone and terminating since we have:

$$dom(\rho_1) = (\{circle\}, \{inside, adjacent\})$$
$$rng(\rho_1) = (Unit, \{par\})$$
$$dom(\rho_2) = (\{circle\}, \{inside\})$$
$$rng(\rho_2) = (Unit, \{body\})$$
$$dom(\rho_3) = (\{circle\}, \{left\text{-}of, adjacent\})$$
$$rng(\rho_3) = (Unit, \{apply\})$$
$$dom(\rho_4) = (\{circle\}, \{left\text{-}of\})$$
$$rng(\rho_4) = (Unit, \varnothing)$$
$$dom(\rho_{VEX}) = (\{circle\}, \{inside, adjacent, left\text{-}of\})$$
$$rng(\rho_{VEX}) = (Unit, \{par, body, apply\})$$

A major source for non-determinism is the competition of different rules for edges of the same label: a pair of rules (r_i, r_j) is *conflicting* if $\pi_2(dom(r_i)) \cap \pi_2(dom(r_j)) \neq \varnothing$. For example, (ρ_1, ρ_2) and (ρ_1, ρ_3) of ρ_{VEX} are conflicting since the intersection of edge labels contains *inside*, respectively, *adjacent*. (In terms of general rewriting systems, (ρ_1, ρ_2) is a critical pair, which destroys confluence in general [13].) Introducing an order among the rewrite rules is in many cases sufficient to recover determinism. To find a reasonable ordering we refine the notion of conflicting rules: we say that r_i *impedes* r_j (denoted by $r_i \angle r_j$) if

$$\exists\, G, G_1, G_2{:}\quad (1)\ G \Rightarrow_{r_i} G_1 \wedge \nexists\, G'{:}\ G_1 \Rightarrow_\rho^\bullet G' \wedge$$
$$(2)\ G \Rightarrow_{r_j} G_2 \wedge \exists\, G'{:}\ G_2 \Rightarrow_\rho^\bullet G'$$

This means that application of r_i prevents G from being completely rewritten to an abstract graph whereas this would be possible by applying r_j instead.

Impediment affects confluence, but not e-confluence, however, the function ρ^* might become less defined since some visual graphs cannot be completely rewritten anymore. If there are no cyclic *impedes* dependencies among rules, we can always apply impeded rules before their impeding ones (that is, order the rule system topologically w.r.t. $\angle$). This means:

Proposition 1. *If the transitive closure of* $\angle_\rho$ *is irreflexive, the domain of* ρ^* *is not affected by conflicting rules.*

For instance, in ρ_{VEX} we have $\rho_2 \angle \rho_1$ and $\rho_4 \angle \rho_3$. (Since *left-of* and *inside* exclude each other, there is no $\angle$ relationship between and ρ_3 and ρ_1.) Thus, $\angle$ is acyclic, and the given numbering gives already a topological sort of the rules. Note that using a rewrite system by trying the rules in such an order makes the rewriting process deterministic (up to the choice of matches).

5. Visual semantics

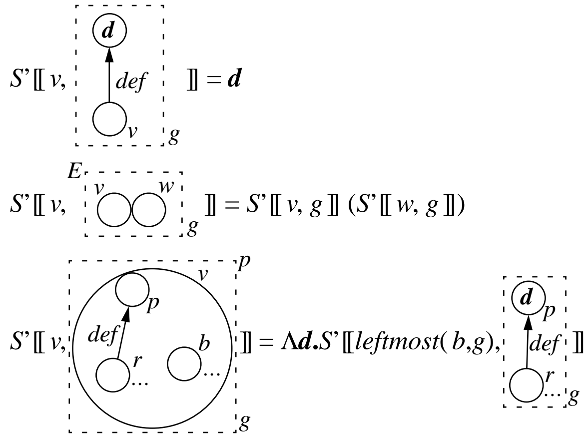
To use visual graphs in semantics definitions we have to fix visual notations for patterns, etc. We use the following conventions: node identifiers are placed next to nodes, respectively, next to the figures representing nodes, and (non-geometric) labels are drawn inside nodes (figures).

In addition to objects of node contexts we need a visual representation of the "rest-graph" allowing to generate a corresponding binding to be used in graph expressions on right hand sides. We use a dotted frame around the visual

graph and annotate it at the lower right corner with the graph variable. We place an *E* at the upper left corner to denote an edge pattern. Otherwise, a node pattern is assumed.

We also need a notation to distinguish between matching single nodes (*lab›v*) and a list of nodes (*lab»v*) with regard to a specific relationship. By default we consider symbols in patterns to represent single nodes. In contrast, nodes annotated by an ellipsis "…" are interpreted as list variables.

Finally, we must be able to specify the depth and order of cascading patterns: by default, an externally bound variable, say *v*, is used to denote a simple node (or edge) pattern *N* (…, *v*, …) *g*. By adding a list of nodes $v_1, …, v_n$ to the upper right corner a cascading node pattern *N* (…, *v*, …) (…, v_1, …) … (…, v_n, …) *g* is denoted. Of course, $v_1, …, v_n$ must be present in the pattern and v_{i+1} must be in relationship to v_i in order to be properly bound. Now we can give a visual semantics for VEX:



This definition differs just in the visual presentation from that of Section 2.3 and is much closer to the original visual language than the textual graph notation. This definition is more readable and easier to understand since it keeps some of the original visual relationships instead of mapping everything to an abstract graph representation.

6. Picture Transformations

It is not only the better readability of definitions that is gained by employing visual graphs. Maybe even more important are applications that repeatedly have to use graph transformations: these are very error-prone and also extremely difficult to follow.

Assume, for example, that we want to formally derive the meaning of the VEX picture from Figure 1. Let us first consider how this works when using textual, abstract syntax graphs. We use the following intermediate graphs:

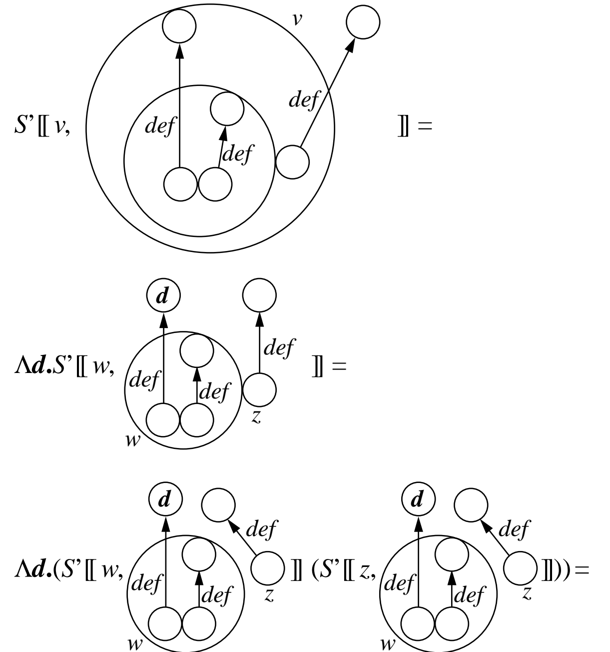
g_1 = *N** (*w*, [*par›a*, *body*»[*y*, *x*], *apply›z*])
(*z*, [*def›c*]) (*c*) (*y*, [*apply›x*]) (*x*, [*def›a*]) (*a*) *Empty*
g_2 = *N** (*w*, [*par›a*, *body*»[*y*, *x*]]) (*z*, [*def›c*]) (*c*)
(*y*, [*def›b*, *apply›x*]) (*b*:***d***) (*x*, [*def›a*]) (*a*) *Empty*
g_3 = *N** (*z*, [*def›c*]) (*c*)
(*y*, [*def›b*, *apply›x*]) (*b*:***d***) (*x*) *Empty*
g_4 = *N** (*y*, [*def›b*]) (*b*:***d***)
(*z*, [*def›c*]) (*c*) (*x*, [*def›a*]) (*a*:***d'***) *Empty*
= *N** (*x*, [*def›a*]) (*a*:***d'***)
(*y*, [*def›b*]) (*b*:***d***) (*z*, [*def›c*]) (*c*) *Empty*

Now we can formally derive:

S'⟦ *v*, *N** (*v*, [*par›b*, *body*»[*w*, *z*]]) ([*y›def*], *b*) g_1 ⟧ =
Λ*d*.*S*'⟦ *w*, *N* ([*y›def*], *b*:***d***, []) g_1 ⟧ =
Λ*d*.*S*'⟦ *w*, *E* (*w›apply›z*) g_2 ⟧ =
Λ*d*.(*S'*⟦ *w*, g_2 ⟧ (*S'*⟦ *z*, g_2 ⟧)) =
Λ*d*.(*S'*⟦ *w*,
*N** (*w*, [*par›a*, *body*»[*y*, *x*]]) ([*x›def*], *a*) g_3 ⟧ ⊥) =
Λ*d*.(Λ*d'*.*S*'⟦ *y*, *N* ([*x›def*], *a*:***d'***, []) g_3 ⟧ ⊥) =
Λ*d*.(Λ*d'*.*S*'⟦ *y*, *E* (*y›apply›x*) g_4 ⟧ ⊥) =
Λ*d*.(Λ*d'*.(*S'*⟦ *y*, g_4 ⟧ (*S'*⟦ *x*, g_4 ⟧)) ⊥) =
Λ*d*.(Λ*d'*.(*d d'*) ⊥) = **Λ*d*.*d*** ⊥

The transformation is difficult to follow, in particular, the need[1] for inventing graph expressions for intermediate graphs makes the whole derivation rather irksome.

In contrast, the same derivation can be performed much easier by using visual graphs:



1. This is inherent in the concept of active patterns since they have to perform non-trivial computations on matched data type values [5].

$\Lambda \boldsymbol{d}.(\Lambda \boldsymbol{d'}.S'[\![y,$ [graph: nodes d, d', x, y, z; edges labelled def] $]\!] \perp) =$

$\Lambda \boldsymbol{d}.(\Lambda \boldsymbol{d'}.(S'[\![y,$ [graph: nodes d, d', x, y, z; edges labelled def] $]\!]\ (S'[\![x,$ [graph: nodes d, d', x, y, z; edges labelled def] $]\!])) \perp) =$

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

7. Related work

The idea of visual graphs is not entirely new: Harel's higraphs [10] are a specific kind of visual graphs. Higraphs are essentially a visual language for application modeling, however, they have a fixed semantics definition which limits their use as a general visual language representation.

Visual manipulation of visual languages has received much attention [2, 3, 4, 9, 11, 12]. All these approaches do work on concrete pictures and not on abstract graph representations. The purely visual treatment is very attractive, but, for example, semantics are sometimes difficult to define and, in particular, difficult to apply and to deal with in proofs. It should also be noted that semantics definitions must always be changed when the syntax of the language changes.

The distinction between concrete and abstract visual syntax has also been proposed in [1, 14]. Since that approach is mainly concerned with the implementation of visual languages, a one-to-one correspondence between concrete and abstract syntax is required, and thus abstract syntax is intrinsically coupled very closely to concrete syntax. This restricts the abstractions that can be achieved by choosing abstract syntax and prevents in some cases semantics definitions altogether. The approach of semi-abstract syntax is more flexible: we can decide how far concrete syntax is preserved and where we recourse to abstract graph edges when it is more convenient.

The inductive view of graphs seems to be closely related to the well-established grammatical approaches to visual languages. Graph grammars, for example, offer also 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 is certainly that graph grammars have a non-trivial semantics themselves (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 contrast, the applicative, data type-like approach proposed in [6, 8] and in this paper is more versatile: we can easily deal with nested graphs (an example is given in [8]), and we can also employ multiple graphs (see the Turing machine example in [7]).

8. Conclusions

We have introduced visual graphs as a formal but visual representation for visual languages. By defining specialized visual rewrite systems the use of visual graphs in formal definitions becomes possible. Together with a structured processing of graphs that is offered by the underlying inductive graph model we are able to formally work with visual languages in an intuitive and easily understandable way. This should make visual language formalisms accessible to a broader audience.

9. References

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

[2] Bell, B. & Lewis, C.: ChemTrains: A Language for Creating Behaving Pictures, *VL*'93, 188-195.

[3] Citrin, W., Doherty, M. & Zorn, B.: Formal Semantics of Control in a Completely Visual Programming Language, *VL*'94, 294-301.

[4] Citrin, W., Hall, R. & Zorn, B.: Programming with Visual Expressions, *VL*'95, 208-215.

[5] Erwig, M.: Active Patterns, *8th Int. Workshop on Implementation of Functional Languages*, 1996, LNCS 1268, 21-40.

[6] Erwig, M.: Functional Programming with Graphs, *2nd Int. Conf. on Functional Programming*, 1997, 52-65.

[7] Erwig, M.: Visual Semantics – Or: What You See is What You Compute, *VL*'98, 96-97.

[8] Erwig, M.: Abstract Syntax and Semantics of Visual Languages, *JVLC 9*, 1998, 461-483.

[9] Furnas, G.W.: New Graphical Reasoning Models for Understanding Graphical Interfaces, *CHI*'91, 71-78.

[10] Harel, D.: On Visual Formalisms, *CACM, Vol. 31*, No. 5, 1988, 514-530.

[11] Kahn, K.M. & Saraswat, V.A.: Complete Visualizations of Cuncurrent Programs and Their Executions, *VL*'90, 7-15.

[12] McIntyre, D.W. & Glinert, E.: Visual Tools for Generating Iconic Programming Environments, *VL*'92, 162-168.

[13] Plump, D.: Hypergraph Rewriting: Critical Pairs and Undecidability of Confluence, in M.R. Sleep, M.J. Plasmeijer & M.C.J.D. van Eekelen (eds.): *Term Graph Rewriting – Theory and Practice*, John Wiley & Sons, 1993, 203-213.

[14] Rekers, J. & Schürr, A.: A Graph Grammar Approach to Graphical Parsing, *VL*'95, 195-202.