In Proc. Human-Centric Computing, IEEE Press, Sep. 2001, 30--37

Shapely Hierarchical Graph Transformation*

Berthold Hoffmann (hof@tzi.de)
Technologiezentrum Informatik, Universität Bremen
Postfach 33 04 40, 28334 Bremen

Abstract

Diagrams can be represented by graphs, and the animation and transformation of diagrams can be modeled by graph transformation. This paper studies extensions of graphs and graph transformation that are important for programming with graphs:

- *We extend graphs by a notion of hierarchy that supports value composition, and define* hierarchical graph transformation *in an intuitive way that resembles term rewriting.*
- *We require that admissable shapes for hierarchical graphs are specified by context-free graph grammars, in order to set up a type discipline for* shapely hierarchical graph transformation.

The resulting computational model shall be the basis of the visual language DIAPLAN *for programming with graphs that represent diagrams.*

1 Introduction

The Unified Modeling Language (UML [21]) is a prominent evidence for the steadily increasing importance of visual languages in computer science. The syntax of a visual language can be represented by *graphs*, and *graph transformation* can be used as a computational model for specifying its semantics [1]. Such a specification can also be used for implementing the visual language [15, 12].

In this paper, we investigate the computational model of a visual language for *programming with diagrams*. We start from a definition of graph transformation that is based on variable substitution [18], and is particularly simple and intuitive. We extend it by two concepts that are fundamental for programming:

- The *composition* of values from values (also recursively) is central for structuring data in programming languages. We extend graphs, along the lines of [7], by a *hierarchy* that forbids edges between components. Other notions of hierarchical graphs that are used for software modeling [3, 9] sacrifice compositionality by allowing edges that cross component boundaries.
- *Typing* of values and operations is important for detecting inconsistencies in programs. We require that the *shapes* of graphs are specified by context free graph grammars [6], and refine transformation so that only shaped hierarchical graphs are manipulated. This goes beyond related typing concepts for graphs, which merely restrict the labelling and degree of nodes and edges [23].

This paper is structured as follows: In sections 2 and 3 we recall substitutive transformation of (plain) graphs. Section 4 introduces hierarchical graphs, to which substitutive transformation is extended in section 5. In section 6, we describe the specification of graph shapes, and refine transformation in section 7 so that it adheres a shape discipline. We conclude with some remarks on related and future work (in section 8).

Acknowledgments. The author thanks Annegret Habel and Frank Drewes for useful comments.

2 Plain Graphs

We base our definitions on graphs that are general enough to express relations of any arity. Graphs may contain variables as placeholders for (sub-) graphs.

Hypergraph. Let Λ be a set of *labels* where every label $l \in \Lambda$ has a *rank* number $\mathrm{rank}(l) \geq 0$. A *hypergraph* over Λ is a quadruple $G = \langle V, E, \mathrm{lab}, \mathrm{att}\rangle$ consisting of finite sets V of *nodes* and E of *edges*, a *labelling function* $\mathrm{lab} : E \to \Lambda$, and an *attachment function* $\mathrm{att} : E \to V^*$ that assigns sequences of nodes to hyperedges so that $\mathrm{att}(e)$ has the length $\mathrm{rank}(\mathrm{lab}(e))$.[1] An edge e with rank k and label l may be qualified as *k-ary*, and as an *l-edge*. From now on, a hypergraph G is simply called a *graph*, and its components are often denoted by V_G, E_G, att_G, and lab_G.

For some label $l \in \Lambda$, $l^\bullet$ shall denote the *handle graph* of l that consists of a single l-edge e that is attached to $\mathrm{rank}(l)$ pairwise disjoint nodes.

Variables. We assume that Λ contains a distinguished set $\mathcal{X}$ of *variable names*. The variable names occurring in a

[1] V^* denotes the set of *sequences* $v_1 \ldots v_k$ over some vocabulary V ($v_i \in V$ for $1 \leq i \leq k$). The *empty sequence* is denoted by ε.

* This work has been partially supported by the ESPRIT Working Group *Applications of Graph Transformation* (APPLIGRAPH).

graph G are denoted by $\mathcal{X}(G)$. Edges labelled by variable names are called *variable edges* (*variables*, for short). A variable e is *straight* in a graph G if its attachments are pairwise distinct. By $\underline{G}$ we denote the *skeleton* of a graph G where all variables have been removed.

Example 1 (Chain Graphs) Our running example is concerned with representing *chains* of graph values. In plain graphs, the chain structure is represented by two kinds of edges: A c-edge is attached to the *begin* and *end* node of a chain of i-edges; every i-edge is in turn attached to the *begin* and *end* node of an *item graph* that is stored in the chain.

Figure 2 shows graphs that contain chain and item graphs. Graphs are drawn in boxes, with their names written in the upper right corner. Nodes are drawn as circles. Edges are drawn as boxes around their label, and are connected to their attachments by lines that are ordered counterclockwise, starting at noon. Variable names (occurring in figure 1) are written in upper case whereas other edge labels appear in lower case. The boxes for binary edges with empty labels "disappear" so that they are drawn as lines from their first to their second attached node (as in "ordinary" graphs). We apply this convention to the item graphs stored in a chain graph.

Morphisms. Structure-preserving node and edge mappings between graphs are used to identify (copies of) a graph as a subgraph in another one.

Let G and H be graphs. A *morphism* $m : G \to H$ is a pair $\langle m_V : V_G \to V_H, m_E : E_G \to E_H \rangle$ of functions that preserve labels and attachments:

$$\mathrm{lab}_H \circ m_E = \mathrm{lab}_G \text{ and } \mathrm{att}_H \circ m_E = m_V^* \circ \mathrm{att}_G$$ [2]

The morphism m is surjective (injective) if its component mappings are surjective (injective, respectively). If m is surjective and injective, it is called an *isomorphism*, and the graphs G and H are called *isomorphic*, denoted by $G \cong_m H$. (We omit the index m if it is not relevant.)

Edge Replacement. An edge e in a graph G may be replaced by another graph U by gluing e's attachments with distinguished nodes in U. This operation is central for defining transformations.

A *pointed graph* $\langle G, p \rangle$ is a graph G with a distinguished sequence $p \in V_G^*$ of *points*. The length of p determines G's *rank*, written $\mathrm{rank}(G)$. Often we denote pointed graphs only by their graph component, and refer to their points by p_G. Points are drawn as filled circles, see Example 2 below.

[2]The *composition* $f \circ g$ of two functions $g : A \to B$ and $f : B \to C$ is a function in $A \to C$ given by $f \circ g(x) = f(g(x))$ for all $x \in A$. The extension $f^* : A^* \to B^*$ of a function $f : A \to B$ maps the empty sequence ε onto itself, and a sequence $a_1 \ldots a_k$ onto the sequence $f(a_1) \ldots f(a_k)$.

The *replacement* of an edge e in a graph G by a pointed graph $\langle U, p \rangle$ is defined if $\mathrm{rank}(\mathrm{lab}_G(e)) = \mathrm{rank}(U)$, and proceeds in two steps: (1) construct the disjoint union of G and U, and remove e, (2) glue the corresponding nodes of $\mathrm{att}_G(e)$ and p.

Instantiation. A *substitution pair* $x \mapsto U$ associates a variable name x with a pointed graph U of the same rank wherein all points are pairwise distinct. A *substitution* is a finite mapping

$$\sigma = \{x_1 \mapsto U_1, \ldots, x_k \mapsto U_k\}$$

of substitution pairs, with pairwise distinct variable names, and the *domain* $Dom(\sigma) = \{x_1, \ldots, x_k\}$. $Subst_\Lambda$ denotes the substitutions over $\mathcal{H}_\Lambda$.

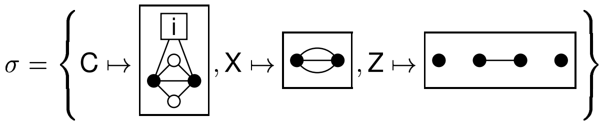
The *instantiation* of a graph P according to a substitution σ is obtained by the parallel replacement of all x-variables e with $x \in Dom(\sigma)$ by their substitution $\sigma(x)$, and is denoted by $P\sigma$. This defines $P\sigma$ uniquely (up to isomorphism) as edge replacement is commutative [6, sect. 2.2.2].

A graph P *matches* a graph G if there is a substitution σ so that $P\sigma \cong G$.

Constructing the instantiation $P\sigma$ for given P and σ is straigthforward. *Graph matching* is far less obvious, but decidable as well, even if this is NP-complete in general (see [18, corollary 15]):

Lemma 1 *It is decidable whether a graph P matches a graph G or not.*

Example 2 (Graph Instantiation) With the substitution



the graphs P_e and R_e in figure 1 match the graphs G and H in figure 2, respectively.

3 Plain Graph Transformation

Graphs are transformed by matching the pattern graph of a rule to a host graph, and rewriting it by the instantiated replacement graph of that rule.

Transformation. A *graph transformation rule* $t = P \to R$ (*rule* for short) consists of a *pattern graph* P and a *replacement graph* R. The rule t *matches* a graph G if there is a substitution σ such that $P\sigma \cong_m G$. The triple $\langle t, m, \sigma \rangle$ is then called a *redex* in G, which *transforms* a graph G into a graph $H \cong R\sigma$, written $G \Rightarrow_t H$.

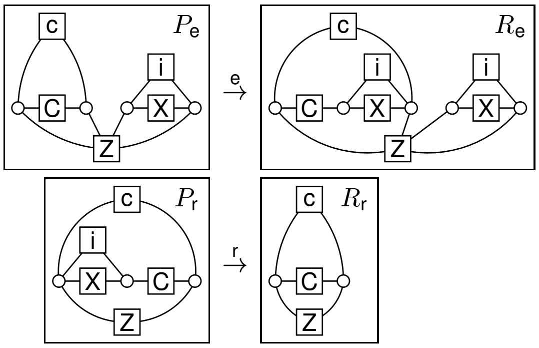


Figure 1. Two plain rules

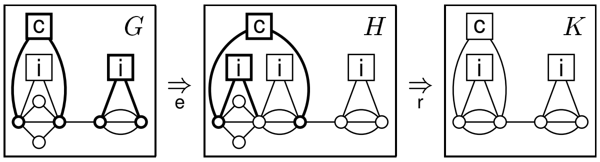


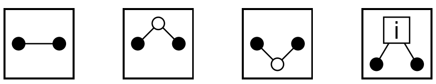
Figure 2. Two plain transformation steps

Example 3 (Chain Graph Transformation) In figure 1, the rule $\mathsf{e} : P_{\mathsf{e}} \to R_{\mathsf{e}}$ *enters* item graphs at the end of a chain graph, and the rule $\mathsf{r} : P_{\mathsf{r}} \to R_{\mathsf{r}}$ *removes* the first item graph of a chain graph. Figure 2 shows two transformation steps using these rules. The substitution σ from example 2 is used in the redex $\langle \mathsf{e}, m, \sigma \rangle$ of the first step. The image of the skeleton $\underline{P_{\mathsf{e}}}$ under the morphism m is highlighted by fat lines in G, and the image of $\underline{P_{\mathsf{r}}}$ under the morphism used in the second step is highlighted in H as well.

By lemma 1, transformation steps are computable. A redex $\langle t, m, \sigma \rangle$ determines the transformed graph uniquely.

Substitutive transformation is more general than the gluing approach [8], since graphs of arbitrary size, namely the substitutions of variables, can be *deleted* or *duplicated* in a single transformation step [18]. In example 3, two occurrences of the variable name X in the graph R_{e} duplicate the item graph $\sigma(\mathsf{X})$ in H, and the ommission of X in the graph R_{r} deletes the item graph substituting X from K.

A rule may have many redices, even in the same context. Instead of the substitution σ of Example 3, any substitution where some of the subgraphs



are removed from $\sigma(\mathsf{C})$, and glued to the two points on the left of $\sigma(\mathsf{Z})$ will also match the pattern P_{e} and the graph G.

This reveals a general deficiency of plain graphs: It is impossible to represent *identifiable* subgraphs, as the chain graph denoted by C, or the item graph denoted by X, in a unique way. This is the reason for introducing a hierarchical structuring to graphs.

4 Hierarchical Graphs

A *hierarchical graph* consists of a plain graph where some edges contain graphs that may be hierarchical again.[3] Our definition follows [7], but allows a wider use of variables.

Hierarchical Graph. The set $\mathcal{H}$ of *hierarchical graphs* consists of sixtuples

$$H = \langle V, E, F, \mathrm{lab}, \mathrm{att}, \mathrm{cts} \rangle$$

where $\widehat{H} = \langle V, E, \mathrm{lab}, \mathrm{att} \rangle$ is a plain graph, called the *top* of H, $F \subseteq E$ is the set of *frame edges* (or just *frames*) that are not labelled by variable names, and $\mathrm{cts} : F \to \mathcal{H}$ is the *contents function* mapping frames to hierarchical graphs.[4] The constituents of a hierarchical graph H are often denoted by V_H, E_H, lab_H, att_H, and cts_H.

Hierarchical Morphism. Let $G, H \in \mathcal{H}_\Lambda$. A *hierarchical morphism* $m : G \to H$ is a pair $m = \langle \hat{m}, M \rangle$ where

- $\widehat{m} : \widehat{G} \to \widehat{H}$ is a plain graph morphism such that $\widehat{m}(f) \in F_H$ for all frames $f \in F_G$, and
- $M = (m_f : \mathrm{cts}_G(f) \to \mathrm{cts}_H(\widehat{m}(f)))_{f \in F_G}$ is a family of hierarchical morphisms between the contents of frames.

As in the plain case, a hierarchical morphism $m : G \to H$ is surjective (injective) if its component morphisms are surjective (injective, respectively). This induces an *isomorphism* relation $G \cong_m H$ if m is surjective and injective.

Component Replacement. Frames are nested in a tree-like hierarchy. Every occurrence of a frame in the hierarchy designates its contents as an isolated (hierarchical) graph component that can be freely replaced by another component.

Let H be a hierarchical graph. For every frame $f \in F_H$, the hierarchical graph $\mathrm{cts}_H(f)$ is denoted by H/f and called a (direct) *component* of H. The *replacement* of the component H/f in H by some hierarchical graph U is the hierarchical graph $H[f \leftarrow U] = \langle V_H, E_H, F_H, \mathrm{lab}_H, \mathrm{att}_H, \mathrm{cts}' \rangle$ with the contents function $\mathrm{cts}'(f') = U$ if $f' = f$, and $\mathrm{cts}'(f') = \mathrm{cts}_H(f')$ otherwise.

We define the *occurrences* in a hierarchical graph H as the frame sequences

$$\Omega_H = \{\varepsilon\} \cup \{fw \mid f \in F_H, w \in \Omega_{H/f}\}$$

[3] Here we have not considered *frame nodes* that may contain hierarchical graphs, as they would not add power to the representation. For, if some node v shall contain a hierarchical graph H_v, H_v may as well be contained in a frame f attached to v.

[4] Because of the recursion in the definition, hierarchical graphs are defined inductively as $\mathcal{H} = \bigcup_{i \geq 0} \mathcal{H}_i$, where $H \in \mathcal{H}_0$ if $F_H = \emptyset$, and, for $i > 0$, $H \in \mathcal{H}_i$ if $\mathrm{cts}_H(f) \in \mathcal{H}_{i-1}$ for every frame $f \in F_H$.

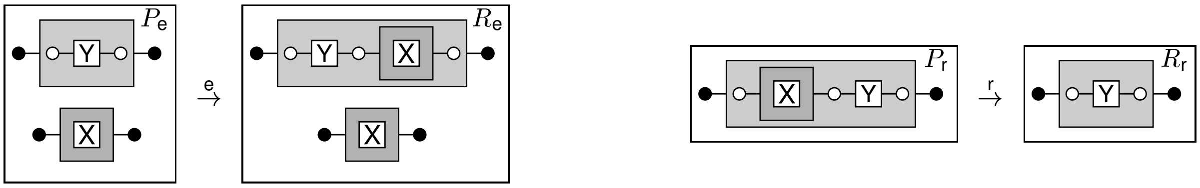


Figure 3. Hierarchical rules for entering (left) and removing (right) items from a chain

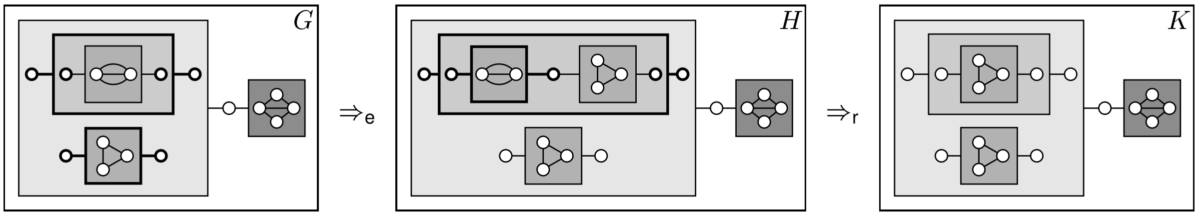


Figure 4. Two hierarchical substitutive transformation steps

and extend the notion of a *component* H/w and of *component replacement* $H[w \leftarrow U]$ to arbitrary occurrences $w \in \Omega_H$ as follows:

$$H/\varepsilon = H, \qquad H/(fw) = (H/f)/w$$
$$H[\varepsilon \leftarrow U] = U, \;\; H[fw \rightarrow U] = H[f \leftarrow (H/f)[w \leftarrow U]]$$

From now on, we silently assume that substitutions and instantiation are extended to hierarchical graphs.

Example 4 (Hierarchical Chain Graphs) For a hierarchical representation of chains, we turn the c- and i-edges of Example 1 into frames that contain chain and item graphs, respectively. Frames are boxes (like ordinary edges), with their contents drawn inside; they are filled in different shades of grey, so that their labels c and i can be omitted.

Figures 3 and 4 shows hierarchical graphs that contain chain graph components. In this representation, chain graphs and item graphs are isolated components. This helps to maintain the consistency of the representation, and allows to compose and decompose them freely.

5 Hierarchical Graph Transformation

It is easy to extend plain graph transformation to hierarchical graphs: In the definition of rules and substitutions, plain graphs are replaced by hierarchical graphs, and a rule may be applied to every component of a hierarchical graph.

We could define a transformation step by requiring that $G/w \cong P\sigma$ for some occurrence w in Ω_G, some hierarchical pattern graph P and some substitution σ, and construct the transformed graph by *component replacement*: $H \cong G[w \leftarrow R\sigma]$. However, this would make it more complicated to impose a shape discipline on transformations in section 7. We therefore define transformation in a slightly different way, by *embedding* pointed pattern and replacement graphs into a *context*, which is a hierarchical graph with a single variable.

Substitutive Hierarchical Transformation. A *hierarchical graph transformation rule* $t : P \rightarrow R$ (*hierarchical rule*, for short) consists of a pointed hierarchical *pattern graph* P wherein all points are pairwise distinct, and a pointed hierarchical *replacement graph* R.

A hierarchical graph is a *context*, and denoted by $C[\,]$, if it contains a single straight *hole variable* named $\triangle$ in one of its components. The *embedding* of a pointed hierarchical graph U with $\mathrm{rank}(U) = \mathrm{rank}(\triangle)$ into $C[\,]$ is defined by replacing the $\triangle$-edge with U, and is denoted by $C[U]$.

A rule $t : P \rightarrow R$ *matches* a hierarchical graph G if there is a context $C[\,]$ and a substitution σ such that $G \cong_m C[P\sigma]$. The triple $\langle t, m, \sigma\rangle$ is then called a *redex* in G, and *transforms* G into another graph $H \cong C[R\sigma]$, denoted by $G \Rightarrow_t H$. Note that every redex of a rule determines the result of rewriting this redex uniquely, up to isomorphism.

Example 5 (Hierarchical Chain Graph Transformation) Figure 3 shows hierarchical versions of the rules e and r in example 3, and transformations using these rules in figure 4. As in figure 2, fat lines indicate the matching of $\underline{P_\mathsf{e}}$ in G and of $\underline{P_\mathsf{r}}$ in H. Figure 5 shows the context and the substitution for the first transformation step.

Hierarchical substitutive transformation is computable, as the hierarchical graph matching problem $C[P\sigma] \cong G$

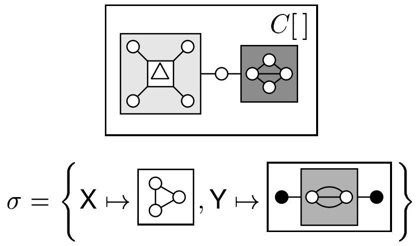


Figure 5. Context and substitution for $G \Rightarrow_{\mathsf{e}} H$

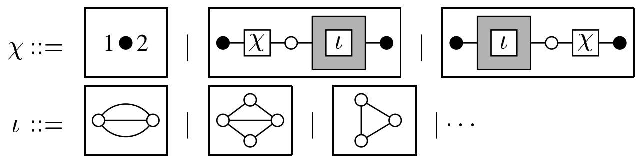


Figure 6. The shape of chain and item graphs

can be solved by matching the finite set $\{\widehat{P/w} \mid w \in \Omega_P\}$ of plain graphs with the corresponding plain graphs in some component of G.

6 Shapes

Shape analysis has been used for inferring whether pointer structures in imperative programs are shaped as doubly-linked lists, root-connected trees and the like [22]. Here we devise a means to *specify* shapes in a way that is not possible on the level of imperative languages (nor in functional or logical languages where pointers are hidden altogether). The specification is based on edge replacement [6], but we immediately adopt a terminology fitting to our purposes.

Shape Grammars. A *shape rule* s has the form $n ::= R$. It associates a *shape name* $n \in \Lambda \setminus \mathcal{X}$ with a variable-free pointed hierarchical graph R. The shape rule s *directly derives* a hierarchical graph G to a hierarchical graph H by replacing one n-edge by R; this is written as $G \Rightarrow_s H$. Shape rules perform edge replacement in hierarchical graphs.

A finite set S of shape rules induces a *shape system* $\Sigma = \langle \mathcal{H}_\Lambda, N, S\rangle$ over the shape names $N \subseteq \Lambda$ that occur as left hand sides in S. Then $\Rightarrow_S$ denotes direct derivation via S, with reflexive-transitive closure $\Rightarrow_S^*$. If $G \Rightarrow_S^* H$, we say that G *derives* H.

For each shape name $n \in N$, the *shape grammar* $\Gamma_n = \langle \mathcal{H}_\Lambda, N, S, n^\bullet\rangle$ with *startgraph* $n^\bullet$ defines the set $\mathcal{F}_n = \{G \in \mathcal{H}_\Lambda \mid n^\bullet \Rightarrow_S^* H\}$ of *shape forms*.

Example 6 (Shapes of Chain Graphs and Item Graphs) Figure 6 defines the shape of chain and item graphs.[5] Shape names are denoted by lower case greek letters, like χ and ι in this case. We write $n ::= R_1 \mid \ldots \mid R_k$ for shape rules $n ::= R_1$ to $n ::= R_k$ with the same shape name.

Shape grammars generate a kind of graph languages that seems to be one of the largest classes that can be called *context-free*. (See [6, section 2.5] for details.) In particular, they allow to define the *recursive algebraic data types* of functional languages that are constructed with disjoint union $+$ and cartesian product $\times$. Moreover, doubly-linked or cyclic lists, leaf- or root-connected trees, and other imperative data structures employing structure sharing can be specified as well [10].

[5]The numbers attached to the node in the first shape of χ expresses that this node is the first *and* second point of an empty chain.

7 *Shapely* Hierarchical Graph Transformation

We now define shaped graphs, and define shapely hierarchical graph transformation so that it preserves shapes.

Shaped Graphs. For the rest of this paper, we fix a shape system $\Sigma = \langle \mathcal{G}_\Lambda, N, S\rangle$, and assume that every variable name $x \in \mathcal{X}$ is *typed* with a shape name $\mathrm{type}(x) \in N$.

The *shape* $\lceil G \rfloor$ of a (pointed) hierarchical graph G is the (pointed) hierarchical graph obtained by relabelling every x-variable with its shape name $\mathrm{type}(x)$, for all $x \in \mathcal{X}(G)$.

The set $\mathcal{G}_\Sigma$ of *shaped graphs* is defined by uniting the sets of *n-shaped graphs* $\mathcal{G}_n = \{G \in \mathcal{G}_\Lambda \mid \lceil G \rfloor \in \mathcal{F}_n\}$ for all $n \in N$. We write $\mathrm{type}(G) = n$ if $G \in \mathcal{G}_n$.

A substitution pair $x \mapsto \langle U, p\rangle$ is *shaped* if $U \in \mathcal{G}_{\mathrm{type}(x)}$. The *shaped substitutions* consist of shaped substitution pairs and are denoted by *Subst*$_\Sigma$.[6]

Lemma 2 *For all hierarchical graphs $G \in \mathcal{G}_\Lambda$ and all shaped substitutions $\sigma \in$ Subst$_\Sigma$:*
(1) $\lceil G \rfloor \Rightarrow_S^* \lceil G\sigma \rfloor$.
(2) $G \in \mathcal{G}_\Sigma$ implies $G\sigma \in \mathcal{G}_\Sigma$ with $\mathrm{type}(G) = \mathrm{type}(G\sigma)$.

Proof. (1) For every shaped substitution pair $x \mapsto U$ in σ, $n^\bullet \Rightarrow_S^* \lceil U \rfloor$ if $n = \mathrm{type}(x)$. If the replacement of some x-variable occurring in G yields a hierarchical graph G', this corresponds to the replacement of an n-edge in $\lceil G \rfloor$ by $\lceil U \rfloor$, yielding $\lceil G' \rfloor$. Then $\lceil G \rfloor \Rightarrow_S^* \lceil G' \rfloor$ follows from the definition of $\Rightarrow_S$. Since edge replacement is associative [6, sect. 2.2.2], we conclude that $\lceil G \rfloor \Rightarrow_S^* \lceil G\sigma \rfloor$.

[6]Note that the pairs in such substitutions may have different shapes.

(2) Let $\lceil G \rfloor \in \mathcal{F}_n$ for some $n \in N$. This means that $n^\bullet \Rightarrow_S^* \lceil G \rfloor$, so that $n^\bullet \Rightarrow_S^* \lceil G\sigma \rfloor$ by transitivity of derivations. Thus $G, G\sigma \in \mathcal{G}_\Sigma$ with $\mathrm{type}(G) = \mathrm{type}(G\sigma) = n$. □

Shapely Transformation. From now on, we restrict our attention to the transformation of shaped graphs by shapely rules.

A rule $t : P \to R$ is *shapely* if its pattern and replacement graphs P and R are shaped so that $\mathrm{type}(P) = \mathrm{type}(R)$. Such a rule *matches* a shaped graph G if there is a shaped context graph $C[\,] \in \mathcal{G}_\Sigma$ with $\mathrm{type}(\triangle) = \mathrm{type}(P)$ and a shaped substitution σ such that $G \cong C[P\sigma]$. Then t *transforms* G to the hierarchical graph $H \cong C[R\sigma]$, written $G \Rrightarrow_t H$.

Lemma 3 *For all shaped context graphs $C[\,] \in \mathcal{G}_\Sigma$ and all pointed shaped graphs U with $\mathrm{type}(U) = \mathrm{type}(\triangle)$, $C[U] \in \mathcal{G}_\Sigma$ with $\mathrm{type}(C[\,]) = \mathrm{type}(C[U])$).*

Proof. Since $C[U] = C[\,]\sigma$ for the (shaped) substitution $\sigma = \{\triangle \mapsto U\}$, lemma 2 applies directly. □

The shape discipline is consistent, since the result of a shapely transformation is a shaped graph again.

Theorem 1 *If $G \Rrightarrow_t H$ by some shapely rule t, then H is an shaped graph whenever G is an shaped graph.*

Proof. Let $\langle t, m, \sigma \rangle$ be the redex such that $G \cong_m C[P\sigma]$. Since $C[\,]$ is a shaped graph by definition, $R\sigma$ and $C[R\sigma]$ are shaped graphs by lemmata 2 and 3. □

In most cases, only variable-free shaped graphs will be transformed. So the question arises whether variable-free shaped graphs are also closed under shapely hierarchical transformation. This is only true if the rules do not introduce new variables in their replacement graphs that do not appear already in their pattern graphs.

Corollary 1 *Let $G \Rrightarrow_t H$ for shaped graphs G, H, and some shapely hierarchical rule t with $\mathcal{X}(R) \subseteq \mathcal{X}(P)$.*
If G is variable-free, then H is variable-free as well.

Proof. If G is variable-free, the substitutions $\sigma(x)$ are variable-free for all $x \in \mathcal{X}(P)$. Since all variables of R are required to occur in P, they are all in $Dom(\sigma)$ so that $R\sigma$ and H are variable-free as well. □

Example 7 (Chain and Item Graph Substitutions) Consider example 5 and the shapes specified in figure 7. Assume that the variable names in Figures 3 and 5 have the types $\mathrm{type}(\mathsf{X}) = \iota$, $\mathrm{type}(\mathsf{Y}) = \chi$, and $\mathrm{type}(\Delta) = \delta$. Then the substitution σ is shaped, the graphs P_e and R_e have the type δ, and the context $C[\,]$ has the type γ. By theorem 1,

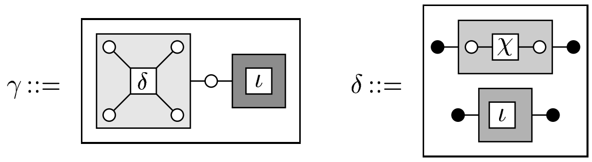


Figure 7. The shape of graphs in Figure 3–5

the graphs G and H are thus (variable-free) shaped graphs of type γ, and the e-step is shapely. (The r-step is shapely as well.)

It is decidable whether some hierarchical graph is shaped or not, since membership in a context-free hypergraph language is decidable [6, sect. 2.7]. Thus shapes set up a practical *shape discipline* that allows to detect errors in hierarchical graphs and hierarchical rules automatically.

Like most other type disciplines that can be *statically* checked (just by inspecting programs), this may also make the application of rules more efficient: The host graph to be transformed can be type-checked by *parsing* it according to its shape grammar. If the shape grammars are *unambiguous* so that every shaped graph has *exactly one* derivation tree, graphs can be represented by these trees, and rule matching boils down to matching a pattern derivation tree to the host graph derivation tree. This would be nearly as efficient as pattern matching in functional languages. (D.A. Watt followed a similar idea when he devised an analysis-oriented restriction of two-level string grammars [24].)

However, the shape of chain graphs as defined in example 6 is ambiguous. It must be, since otherwise either R_e or P_r would fail to be shaped. For ambiguous shapes, the adaptation of the Cocke-Kasami-Younger parsing algorithm to graphs devised in [15] can still be employed, as it parses ambiguous grammars as well. However, the number of derivation trees may be exponential in general. C. Lautemann defined a class of shape grammars where the number of derivation trees is at most polynomial [14].

8 Conclusions

We have extended graphs by a compositional notion of hierarchy, and have devised a notion of hierarchical graph transformation that extends the rewriting of terms [13] (which are trees over function symbols) to trees over graphs (namely, hierarchical graphs) in a straight-forward way: Transformation substitutes variables in a rule pattern, embeds the instantiated pattern into a context, and then inserts an instantiated replacement into that context. This way of graph transformation is quite intuitive from a programming point of view. Furthermore, it can easily be refined by a *shape discipline* that is consistent and decidable, and may also allow for more efficient implementation. Hopefully this is accepted as excuse for inventing yet another variation of graph transformation.

Related Work. T.W. Pratt was probably the first to define hierarchical graph languages [19]. He specified the semantics of programming languages by context-free graph grammars, but did not consider further transformation of these graph languages. Engels and Heckel [9] study hierarchical graph transformation as the basis for system modeling languages like UML [21]. They allow edges betwen components (crossing frame borders), which is necessary in that application domain, but would not be adequate for programming since it would give away (de-)compositionality of hierarchical graphs.

Substitutive hierarchical graph transformation is a (modest) extension of hierarchical graph transformation [7], where variables denote the *entire* contents of frames. With our kind of transformation, it is essential that variables may denote arbitrary subgraphs (see Figure 3).

Busatto *et al.* [3] investigate a generic notion of hierarchical graph transformation by which other approaches can be simulated [2], also that of [7], and probably shapely hierarchical graph transformation as well.

The way we define shapes has been inspired by the work of P. Fradet and D. LeMetayer on *Structured Gamma* [10] that uses structured multiset rewriting, a notion that can be "translated" to graph transformation and edge replacement in a straight-forward way.

Next Steps. Transformation may be highly nondeterministic. This may lead to an overload of backtracking. So, nondeterminism has to be restricted to the degree that is really needed for solving a particular problem. Good design of rule patterns, hierarchical structure, and shapes can already reduce nondeterminism. However, even for a given context and rule, there may be many substitutions σ matching the rule to the host graph (see example 3). An important goal will thus be to bind the nondeterminism imposed by substitution. For instance, one could require that every frame in a pattern contains at most one variable. (The pattern graphs in Figure 3 are of this kind.) This would make rule matching nearly as efficient and deterministic as for the gluing approach in [7]. However, already the search for a pattern skeleton $\underline{P}$ in a graph G (the *subgraph isomorphism problem*) is known to be NP-hard in general.

Shapes are just a "structural" way of classifying values according to their (graph-ical) representation. More type discipline would be useful, for instance as in [9] or in PROGRES [23]. Also, context-free graph languages might be too restricted for specifying the shapes of graphs that occur in certain applications. Then Church-Rosser graph languages [17] could be considered.

The reader may also have noticed that it might be nice to have *polymorphic shapes*: The chain type χ in example 6 should rather be a *type schema* $\chi(\alpha)$ with a *type parameter* α that can be instantiated by concrete types ι, $\chi(\iota)$ and the like. Then the rules e and r could be defined polymorphically for chains containing items of *any* shape, as in a functional language.

The Perspective. Shapely hierarchical graph transformation shall become the computational model for the rule-based language DIAPLAN for *programming with hierarchical graphs* [11]. Further concepts of DIAPLAN shall be defined on top of this model:

- *Transformation predicates* with parameters shall allow to *abstract* from transformation sequences.
- *Application conditions* and *predicate parameters* shall allow to specify *control* in an imperative or functional way.
- *Classes* shall *encapsulate* shape rules and transformation predicates.
- Primitive values like *numbers* and *strings* shall be integrated into the graph model.

Moreover, DIAPLAN shall be integrated with the DIAGEN tool [15] for generating diagram editors, yielding a language and tool for *programming with diagrams* that is itself visual [12]:

- DIAGEN can generate editors for arbitrary diagram languages that are represented as shaped hierarchical graphs. Such editors allow to construct input of DIAPLAN programs, and to display their results, in a diagram notation tailored to the program's application domain.
- DIAPLAN can be used to program the semantics of diagrams in terms of the shaplely hierarchical graphs that represent them internally. These semantic operations can then be called from the user interface of the editors.

For instance, DIAPLAN could then be used to extend the Statecharts editor generated by DIAGEN in [16] with operations that animate the behavior of Statecharts, or with transformations that simplify them.

References

[1] R. Bardohl, M. Minas, A. Schürr, and G. Taentzer. Application of graph transformation to visual languages. In G. Engels, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. II: Specification and Programming*, chapter 3, pages 105–180. World Scientific, Singapore, 1999.

[2] G. Busatto and B. Hoffmann. Comparing notions of hierarchical graph transformation. *Electronic Notes in Theoretical Computer Science*, 2001. to appear.

[3] G. Busatto, H.-J. Kreowski, and S. Kuske. An abstract hierarchical graph data model. Technical report, Fachbereich Mathematik-Informatik, Universität Bremen, to appear 2001.

[4] V. Claus, H. Ehrig, and G. Rozenberg, editors. *Proc. Graph Grammars and Their Application to Computer Science and Biology*, number 73 in Lecture Notes in Computer Science. Springer, 1979.

[5] J. E. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors. *Proc. Graph Grammars and Their Application to Computer Science*, number 1073 in Lecture Notes in Computer Science. Springer, 1996.

[6] F. Drewes, A. Habel, and H.-J. Kreowski. Hyperedge replacement graph grammars. In Rozenberg [20], chapter 2, pages 95–162.

[7] F. Drewes, B. Hoffmann, and D. Plump. Hierarchical graph transformation. *Journal of Computer and System Sciences*, to appear 2001. (A short version appeared in number 1784 of Lecture Notes in Computer Science, pages 98–113, 2000).

[8] H. Ehrig. Introduction to the algebraic theory of graph grammars. In Claus et al. [4], pages 1–69.

[9] G. Engels and R. Heckel. Graph transformation as a conceptual and formal framework for system modelling and evolution. In U. Montanari, J. Rolim, and E. Welz, editors, *Automata, Languages, and Programming (ICALP 2000 Proc.)*, number 1853 in Lecture Notes in Computer Science, pages 127–150. Springer, 2000.

[10] P. Fradet and D. L. Métayer. Structured Gamma. *Science of Computer Programming*, 31(2/3):263–289, 1998.

[11] B. Hoffmann. From graph transformation to rule-based programming with diagrams. In M. Nagl, A. Schürr, and M. Münch, editors, *Int'l Workshop on Applications of Graph Transformations with Industrial Relevance (*AGTIVE*'99), Selected Papers*, number 1779 in Lecture Notes in Computer Science, pages 165–180. Springer, 2000.

[12] B. Hoffmann and M. Minas. A generic model for diagram syntax and semantics. In J. D. P. Polim, A. Z. Broder, A. Corradini, R. Gorrieri, R. Heckel, J. Hromkovic, U. Vaccaro, and J. B. Wells, editors, *ICALP Workshops 2000*, number 8 in Proceedings in Informatics, pages 443–450, Waterloo, Ontario, Canada, 2000. Carleton Scientific.

[13] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.

[14] C. Lautemann. The complexity of graph languages generated by hyperedge replacement. *Acta Informatica*, 27:399–421, 1990.

[15] M. Minas. Concepts and realisation of DIAGEN. *Science of Computer Programming*, to appear 2001.

[16] M. Minas and B. Hoffmann. Specifying and implementing visual process modeling languages with DIAGEN. *Electronic Notes in Theoretical Computer Science*, to appear 2001.

[17] D. Plump. Church-Rosser hypergraph languages. Talk at the Workshop "Automaten und Formale Sprachen", Schauenburg-Elmshagen, Germany, September 1999.

[18] D. Plump and A. Habel. Graph unification and matching. In Cuny et al. [5], pages 75–89.

[19] T. W. Pratt. Definition of programming language semantics using grammars for hierarchical graphs. In Claus et al. [4], pages 389–400.

[20] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations*. World Scientific, Singapore, 1997.

[21] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison Wesley, 1999.

[22] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, 1998.

[23] A. Schürr, A. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In Rozenberg [20], chapter 13, pages 487–550.