Fundamenta Informaticae 1–27
IOS Press

Abstraction and Control for Shapely Nested Graph Transformation*

Berthold Hoffmann
Technologiezentrum Informatik, Universität Bremen
Postfach 330 440, D-28334 Bremen, Germany
hof@tzi.de

Abstract. *Shapely nested graph transformation* is the computational model for DIAPLAN, a language for programming with graphs representing diagrams that is currently being developed. The model supports nested structuring of graphs, structural graph types (*shapes*), and rules with graph variables. We extend this model by two concepts that are essential for programming: *abstraction* allows compound transformations to be named and parameterized; *control* allows the order of rule application to be specified. These concepts refine the computational model with respect to structuring and efficiency needs of a programming language while preserving its rule-based and *graph*-ical nature.

Keywords: Graph, nested graph, diagram, graph typing, graph transformation, abstraction, control

1. Introduction

Rules appear everywhere in computer science, for describing or verifying the behavior of systems in an axiomatic way. In particular, they can be used to define models of computation by *reduction systems* $\langle \mathcal{V}, \curvearrowright \rangle$: Given some set $\mathcal{V}$ of values, rules induce a rewrite relation $\curvearrowright \subset \mathcal{V} \times \mathcal{V}$ by which values $v_0 \in \mathcal{V}$ can be reduced to normal form, i.e. rewritten as long as possible. In general, reduction systems describe nondeterministic computations: a value v_0 may reduce to several distinct normal forms. (And, like any algorithm, the computations may be non-terminating: some reductions of v_0 may not lead to a normal form at all.)

*A short version of this paper appeared in the proceedings of the 1st International Conference on Graph Transformation (ICGT'02), see [31].

Two reduction systems are established as foundations of programming paradigms: *Term rewriting* $\langle \mathbb{T}, \rightarrow \rangle$ is a computational model for functional programming [36].[1] The terms $\mathbb{T}$ are trees over function symbols, the rewrite rules define functions, and term rewriting $\rightarrow$ evaluates these functions, usually assuming that normal forms are unique. *Horn clause resolution* $\langle \mathbb{C}, \vdash \rangle$ is the computational model of logic programming [47]. The Horn clauses $\mathbb{C}$ are multisets of terms with variables, the rules define predicates, and resolution $\vdash$ enumerates substitutions of the variables for which the start clause holds.

Graph transformation $\langle \mathbb{G}, \Rightarrow \rangle$ is another reduction system [48]. Graphs allow values to be represented with *sharing* (or *pointers*), as needed in imperative and object-oriented programming languages, but also for the implementation of functional and logic languages. Graph transformation has thus been used to define implementation-oriented computational models for functional programming (*term graph rewriting* [45]), logic programming [11], and also for object-oriented programming [54].

In this paper, we are interested in another quality of graphs: they have a *visualization* as node-and-line diagrams that can be easily adapted to specific notations as they are used in diagram languages like UML [49]. This qualifies graph transformation as a computational model for *visual programming* [7], of systems that manipulate diagrams, by applying graphical rules [4].

If reduction systems shall be used for programming, the scale of software, and efficiency and safety requirements on programs make it necessary to extend these computational models by concepts for structuring, control, and typing:

- The values should support *data structuring*.
- An *abstraction mechanism* should allow to structure and parameterize reductions.
- *Control mechanisms* should allow to eliminate unwanted nondeterminism in the reductions.
- A *type discipline* should detect inconsistencies in values and rules.
- Values, rules and abstractions should be *encapsulated* in modules.

In this paper, we describe concepts for a language for programming with graphs and diagrams that is based on graph transformation. This is inspired by the way how modern functional programming languages like HASKELL [42] have been designed on top of term rewriting, and modern logic programming languages like OZ [52] have been developed on top of Horn clause resolution.

Our way of graph transformation has been derived from a simple version of double pushout graph transformation [17, 9, 18]. In [16, 30], this model has been extended by the following concepts:

- Graphs are *nested*: their edges may contain graphs that may be nested again.
- Graphs are *shaped*, i.e. they are structured according to context-free graph grammars.
- Rules preserve shapes, and may contain *variables* so that transformation steps may move, delete or duplicate the subgraphs that are bound to these variables.

Thus *shapely nested graph transformation*, the resulting computational model, does already provide for data structuring, by nesting, and for typing, by shapes. Here we extend it by further concepts that are essential for programming:

[1] The other one, even more famous, and also somewhat rule-based, is the lambda calculus [5].

- We propose an abstraction concept that allows to define and parameterize compound transformations as predicates.

- We provide control of the reduction order by an overall strategy (depth-first innermost evaluation), as well as by user-definable completion clauses and applicability conditions.

(*Encapsulation*, another important programming concept, will be only briefly discussed, in the conclusions.) The extension shall be seamless so as to preserve the rule-based and graphical nature of the underlying computational model. It shall provide the foundation for DIAPLAN, a language for programming with graphs that is currently being designed by Frank Drewes (Umeå), Mark Minas (München), and the author [29, 33, 14, 15]. DIAPLAN shall complement DIAGEN [41], Mark Minas' tool for generating editors that handle the *syntax* of diagram languages, by a language and tool for programming the *semantics* of such languages. However, there is no space in this paper for discussing the representation of graphs in customized diagram notation. Concerning this subject, the reader is referred to [32].

The rest of the paper is structured as follows. Sections 2 to 4 recall the ingredients of shapely nested graph transformation: graphs with nesting, types and shapes for graphs, and shapely transformation of nested graphs. This is done as far as it is essential for defining the major programming concepts proposed in the paper: abstraction (in Section 5), and control (in Section 6). In Section 7, we compare these concepts to those in related languages, and outline some further research.

Acknowledgments. I want to thank Frank Drewes, Annegret Habel, Mark Minas, and Detlef Plump for valuable contributions to this work. In particular, Annegret and Detlef have devised graph transformation with variables [46], Detlef has brought the work on graph shapes to my attention, while Frank and Mark have been active and tireless in (seemingly) never-ending discussions about the design of DIAPLAN. Also, 10^3 thanks go to Andrea Corradini and Hans-Jörg Kreowski for offering 3^3 pages to present my ideas at this place.

2. Graphs as Values

Graphs represent structures consisting of entities (nodes) and relationships (edges) so that they can be drawn as diagrams. This has made them popular in computer science and beyond. In particular, graphs can represent all kinds of structured values that are used in programming languages: recursive values like lists and trees, as they occur in functional or logic languages, but also more sophisticated structures that exploit sharing (pointers), like doubly-linked lists or leaf-connected trees, as they occur in imperative or object-oriented languages.

Graphs exist in many variations [6, 27]. Our definition is based on hypergraphs, and extended by a nesting concept. Nodes and edges of a graph may contain graphs, and the nodes and edges of the contained graphs may again contain graphs, in a nested fashion. This is important for representing values so that operations on them can be defined in a structured way.

Hypergraphs. The edges of a hypergraph may connect any number of nodes, not just two. In our definition, based on [25], the nodes connected by an edge are ordered, and several edges may be *parallel* (i.e. connect the same nodes).

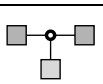
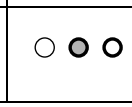
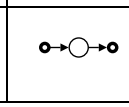
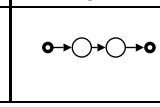
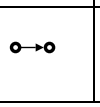
	G_1	G_2	G_3	G_4	G_5	G_6	G_7	G_8
N_i	a	b, c, d	e, f, g	h, i, j, k	l, m	n, o	p, q, r	s, t
E_i	A, B, C		D, E	F, G, H	I	J, K	L, M, N	O, P, Q
a_i	A ↦a B ↦a C ↦a		D ↦ef E ↦fg	F ↦hi G ↦ij H ↦jk	I ↦lm	J ↦no K ↦no	L ↦pq M ↦qr N ↦pr	O ↦st P ↦st Q ↦st
								

Figure 1. Nodes, edges, associations, and diagrams of eight hypergraphs

A *hypergraph* is a triple $G = \langle N, E, a\rangle$ with two disjoint finite sets N and E of *nodes* and *edges* respectively, and a function $a\colon E \to N^*$ that defines the *association* of edges to nodes.[2] The set $N \cup E$ is called the *object set* of a hypergraph G.

Let $G = \langle N, E, a\rangle$ and $H = \langle N', E', a'\rangle$ be two hypergraphs. Then G is a *subhypergraph* of H, written $G \subset H$, if $N \subset N'$, $E \subset E'$, and a is the restriction of a' to E.

A function m from the object set $N \cup E$ to the object set $N' \cup E'$ is a *hypergraph morphism*, written $m\colon G \to H$, if it maps nodes onto nodes, edges onto edges, and preserves the associations of edges, i.e.:

$$\begin{array}{rcll} m(o) \in N' & \Leftrightarrow & o \in N & \text{for all objects } o \in N \cup E \\ a'(m(e)) & = & m^*(a(e)) & \text{for all edges } e \in E \end{array}$$ [3]

A hypergraph morphism is injective (surjective, bijective) if its underlying function has this property, and two hypergraphs G and H are *isomorphic*, written $G \cong H$, if there is a bijective hypergraph morphism $m\colon G \to H$.

We do not distinguish isomorphic hypergraphs. Thus the class $\mathbb{H}$ of *hypergraphs* consists of "abstract hypergraphs", i.e. isomorphism classes of hypergraphs. Nevertheless, definitions, constructions, and results will often be formulated for "concrete" hypergraphs, which should then be considered as representatives of their isomorphism classes.

Example 2.1. (Hypergraphs)
Figure 1 defines the node sets, edge sets, and the association functions of hypergraphs G_1 through G_8, and shows their diagrams in the bottom row. We depict hypergraphs as follows: Nodes are drawn as circles or ovals, and a hyperedge e with association sequence $a_i(e) = n_1 \cdots n_k$ ($k \geqslant 0$) is drawn as a box that is connected with k lines from e to its associated nodes $n_1 \ldots n_k$. Hypergraphs are thus drawn like the corresponding bipartite graphs. By default, the lines connected to e are assumed to be ordered counter-clockwise, starting at noon. Otherwise, the position of a node in e's association sequence is indicated by attaching the position i to the line connecting e with n_i. If a hyperedge e is associated to $k = 2$ nodes, it may be drawn as an arc from n_1 to n_2 (i.e. as in a directed graph). According to this

[2] A^* denotes the set of *finite sequences* over some set A, with the *empty sequence* ε.
[3] If $f\colon A \to B$ is a function, then $f^*\colon A^* \to B^*$ denotes its extension to sequences defined by $f^*(a_1 \cdots a_n) = f(a_1) \cdots f(a_n)$ for $a_i \in A$, $1 \leqslant i \leqslant n$, $n \geqslant 0$.

convention, the hypergraphs G_2 through G_8 can be drawn like directed graphs. (The significance of node and edge shading and line width will be explained when we discuss typing in Subseection 3.1).

Nesting. In [16, 30], hypergraphs have been extended so that their edges may contain hypergraphs, the edges of which may again contain hypergraphs, recursively. Here, we allow that nodes may contain hypergraphs in the same nested fashion. This is more regular, and useful when we define predicates in Section 5. For simplicity, the resulting structures are simply called "graphs".

The class $\mathbb{G}^d$ of *graphs of nesting depth* d ($d \geqslant 0$) consists of quintuples $G = \langle N, E, a, C, (G\mathbf{.}\, c)_{c\in C}\rangle$ where $\langle N, E, a\rangle$ is a hypergraph and the following holds: If $d = 0$ then $C = \emptyset$; otherwise, $C \subset N \cup E$ and $G\mathbf{.}\, c \in \mathbb{G}^{d-1}$ for all $c \in C$. Notice that $\mathbb{G}^d \subset \mathbb{G}^{d+1}$ for $d \geqslant 0$. For, $\mathbb{G}^0 \subset \mathbb{G}^1$, as $C = \emptyset$ trivially implies that G belongs to $\mathbb{G}^1$; then $\mathbb{G}^d \subset \mathbb{G}^{d+1}$ follows by an obvious induction on d. The class of all graphs is $\mathbb{G} = \bigcup_{d\geqslant 0} \mathbb{G}_d$. Given a graph $G = \langle N, E, a, C, (G\mathbf{.}\, c)_{c\in C}\rangle$, we let $\hat{G}$ denote its *top hypergraph* $\langle N, E, a\rangle$. A node or edge in $c \in C$ is called a *container*, and the graph $G\mathbf{.}\, c$ is called its *contents*; all other nodes and edges are called *atomic*.

We extend the notation for designating contents to arbitrary nesting depth in a graph G, by defining the *container positions* in G as the sequences

$$\Gamma_G = \{\varepsilon\} \cup \{c\,\gamma \mid c \in C, \gamma \in \Gamma_{G\mathbf{.}c}\}$$

The top position $\varepsilon \in \Gamma_G$ designates the entire graph, i.e. $G\mathbf{.}\,\varepsilon = G$, and a nested container position $c\gamma \in \Gamma_G$ designates the graph $G\mathbf{.}\, c\gamma = (G\mathbf{.}\, c)\mathbf{.}\,\gamma$. Furthermore, $G(\gamma)$ shall denote the top hypergraph of the graph $G\mathbf{.}\,\gamma$ at a position $\gamma \in \Gamma_G$. Thus a graph G can be considered as a mapping from container positions $\gamma \in \Gamma_G$ to hypergraphs $G(\gamma)$. It is no coincidence that this is similar to definitions of terms as mappings of tree positions to function symbols [34]: graphs with nesting are *trees of hypergraphs*.

Let $G = \langle N, E, a, C, (G\mathbf{.}\, c)_{c\in C}\rangle$ and $H = \langle N', E', a', C', (H\mathbf{.}\, c)_{c\in C'}\rangle$ be graphs. Then G is a *(nested) subgraph* of H, written $G \Subset H$, if H has a container position $\gamma \in \Gamma_H$ such that $\hat{G} \subset H(\gamma)$, every container $c \in C$ is a container in $H(\gamma)$ and $G\mathbf{.}\, c \Subset H\mathbf{.}\,\gamma c$, recursively. A *graph morphism* from G to H is a pair $\langle \hat{m}, M\rangle$ such that

- $\hat{m}\colon \hat{G} \to \hat{H}$ is a hypergraph morphism that preserves containers, i.e. $\hat{m}(c) \in C'$ for every $c \in C$, and
- $M = (m_c\colon G\mathbf{.}\, c \to H\mathbf{.}\,\hat{m}(c))_{C\in C_G}$ is a family of graph morphisms.

A graph morphism is denoted as $m\colon G \to H$; it is injective (surjective, bijective) if all hypergraph morphisms in m have the respective property. Two graphs G and H are *isomorphic*, written $G \cong H$, if there is a bijective graph morphism $m\colon G \to H$. Then m is called a *graph isomorphism*. As with hypergraphs, we do not distinguish isomorphic hypergraphs; every "concrete" hypergraph should be considered as a representative of its isomorphism class.

Example 2.2. (Graph)
We define eight graphs H_1 through H_8 so that their top hypergraphs are the hypergraphs G_1 through G_8 defined in Example 1. Figure 2 defines the containers and contents graphs of H_1 to H_4; the remaining graphs, H_5 to H_8, do not have containers, and are of depth 0. Thus H_3 and H_4 are of depth 1, and H_2 is of depth 2. Finally, H_1 is of depth 3 and has the container positions $\Gamma_{H_1} = \{\varepsilon, \mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{Bb}, \mathrm{Bc}, \mathrm{Bd}, \mathrm{Cf}, \mathrm{Bci}, \mathrm{Bcj}\}$.

	H_1	H_2	H_3	H_4
C_i	A, B, C	b, c, d	f	i, j
$G.c$	A $\mapsto H_5$ B $\mapsto H_2$ C $\mapsto H_3$	b $\mapsto H_6$ c $\mapsto H_4$ d $\mapsto H_7$	f $\mapsto H_8$	i $\mapsto H_8$ j $\mapsto H_7$

Figure 2. Table defining four graphs

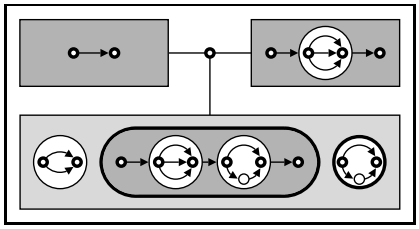


Figure 3. Diagram of the graph H_1

Figure 3 shows the diagram of H_1; it illustrates our conventions for drawing graphs. The circles, ovals and boxes of container nodes and edges are blown up so that their contents fits inside.

We have mentioned that hypergraphs can already represent any kind of structured values. Nevertheless, the secondary structure is important for manipulating graphs, because this allows to put parts of a graph into containers, and move, delete, or duplicate them, or to apply operations to them as if they were just nodes or edges. Without containers, parts of a graph can only be designated as subgraphs, e.g. by marking their nodes and edges, in order to do such manipulations. This, however, would be a rather low-level way of programming.

The *hierarchical graphs* used for system modeling [8, 21] provide another secondary structure for graphs: the nodes and edges of some underlying graph (like a hypergraph) are grouped in packages; these packages may form a hierarchy and have interfaces that designate some of their nodes and edges as "public". Several definitions of hierarchical graphs allow that packages *share* subgraphs, or that edges cross package borders. Then a single package can no longer be manipulated independently from the others, because the removal of a node or edge may affect other packages sharing it, and might invalidate border-crossing edges pointing to a removed node. In contrast to that, our nesting concept is *compositional*: Sharing and border-crossing edges are forbidden so that the contents of every container in a graph can be manipulated, independently of the other ones. This property is essential for programming.

3. Graph Types and Shapes

Often, not every graph in the class $\mathbb{G}$ is considered as a consistent value for some program:

- Nodes, edges, and graphs as a whole may be classified into different *types*, and required to be used according to this classification.
- Graphs may be required to have a certain structure, or *shape*.

We first introduce *typed pointed graphs*, for classifying nodes, edges, and graphs themselves, and then describe *shape rules* by which the structure of graphs can be specified.

3.1. Types and Points

We define a simple way to classify the nodes and edges of graphs, by morphisms [10], and classify graphs in a similar way, by the number and types of distinguished nodes.

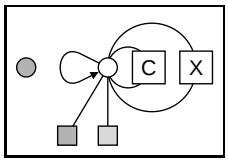


Figure 4. A type hypergraph

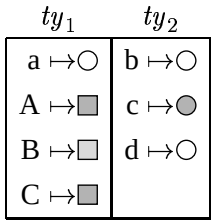


Figure 5. Type morphisms for G_1 and G_2 in example 2.1

A *type hypergraph* $\mathcal{C}(X) = \langle \mathcal{C}_N, \mathcal{C}_E \cup X, a \rangle$ is a hypergraph that contains three mutually disjoint sets $\mathcal{C}_N$ of *node constant names*, $\mathcal{C}_E$ of *edge constant names*, and X of *variable names*. In a type hypergraph, the function a defines the *arity* of edge constants and variables.

A *typed pointed hypergraph over* $\mathcal{C}(X)$ is a triple $\langle G, ty, p \rangle$, consisting of a hypergraph $G = \langle N, E, a \rangle$, a hypergraph morphism $ty\colon G \to \mathcal{C}(X)$, and a sequence $p \in N^*$ of distinguished nodes, called *points*. $\mathbb{H}_{\mathcal{C}}(X)$ denotes the class of abstract typed pointed hypergraphs. Mostly, we denote a typed pointed hypergraph by the hypergraph G alone, and refer to its type morphisms and point sequence by ty_G and p_G, respectively.

Let $G \in \mathbb{H}_{\mathcal{C}}(X)$. Objects $o \in$ in G with $ty_G(o) \in \mathcal{C}_N \cup \mathcal{C}_E$ are called *constant nodes* or *constant edges*, respectively, and all other edges are called *variable edges* (*variables* for short.) $Var(G) \subset X$ shall denote the *variable names* occurring in G. Analogously to edges, hypergraphs may be classified according to the number and types of their points. Thus we define $a(G) = ty^*_G(p_G)$ to be the *arity* of a typed pointed hypergraph G.

Let $G, H \in \mathbb{H}_{\mathcal{C}}(X)$. A hypergraph morphism $m\colon G \to H$ is a *typed hypergraph morphism* if $ty_G(o) = ty_H(m(o))$ for all objects o in G. Two typed, pointed G and H are *isomorphic* if there is an injective and surjective typed hypergraph morphism m that preserves points, i.e. $m^*(p_G) = p_H$.

The class $\mathbb{G}_{\mathcal{C}}(X)$ of *typed pointed graphs* is constructeded from $\mathbb{H}_{\mathcal{C}}(X)$ analogously as $\mathbb{G}$ is constructed from $\mathbb{H}$. Thus every container position $\gamma \in \Gamma_G$ in a typed pointed graph $G \in \mathbb{G}_{\mathcal{C}}(X)$ contains a typed pointed hypergraph $G(\gamma) \in \mathbb{H}_{\mathcal{C}}(X)$. We require that the variables in a typed pointed graph are atomic. Typed pointed graph morphisms, and isomorphism of typed pointed graphs is also defined in analogy to graph morphisms.

If G is a typed pointed graph, its variable names are defined as $Var(G) = \bigcup_{\gamma \in \Gamma_G} Var(G(\gamma))$, and its *arity* is defined as $a(G) = a(\hat{G})$. The subgraph of G where all variable edges are removed is called the *skeleton* of G, and denoted by $\underline{G}$. G is a *ground graph* if $Var(G) = \emptyset$ (and $G = \underline{G}$). The class $\mathbb{G}_{\mathcal{C}}$ of *ground graphs* represents values manipulated by a program, while variables are placeholders for graphs that only occur in rules and programs.

Example 3.1. (Type Hypergraph)
Figure 4 shows a type hypergraph for the hypergraphs and graphs occurring in this paper. Constant node and edge types are distinguished by their shading, and variable types bear names that are drawn inside their boxes.

Figure 5 defines the type morphisms for the graphs G_1 and G_2 in Figure 1; in the remaining graphs G_3 to G_8, all nodes have type ○, and all edges have type →, respectively.

In typed and pointed hypergraphs and graphs, we draw nodes and edges as in the type hypergraph.

That is, types are drawn like *colors*, or *labels*, and the type hypergraph serves as a *schema* for drawing nodes and edges in graphs.

Points are distinguished by drawing their circles and ovals with fat lines. The chain graphs in Figure 3 have two points, their begin and end nodes. Analogously to associations, the points of a hypergraph are ordered counter-clockwise, starting at noon.

Assumption 3.1. We fix a type hypergraph $\mathcal{C}(X)$ for the rest of this paper.

We shall assume all graphs to be typed and pointed. So we mostly call them just graphs, omitting the qualifications "typed" and "pointed".

To avoid unnecessary technicalities, we adopt the natural and practically harmless restriction for hypergraphs that the association sequences of their edges do *not* contain repititions, and that the same holds for their point sequences. For obvious reasons, this assumption does not hold for the type hypergraph $\mathcal{C}(X)$, see Figure 4.

3.2. Shapes

The term *shapes* has come into use for pointer-based data structures, like doubly-linked or cyclic lists, root-connected of leaf-connected trees. Such shapes frequently occur in imperative programs, but cannot be described precisely by type declarations in imperative languages [50].

Graphs like the chain and item graphs occurring in Figure 3 do also have particular shapes that go beyond typing their nodes and edges. So we want to specify the shape of graphs as well. We first discuss general requirements on shapes, and then consider in more detail how context-free shapes can be specified.

Compositional Shapes for Graphs. Let us consider shapes of hypergraphs first. This is comparable to the shape of one pointer structure in isolation.

We assume that some *shape rules* Σ (of a kind that is irrelevant for the moment) allow to classify hypergraphs $\mathbb{H}_{\mathcal{C}}(X)$ according to a finite set S of *shape names*. We write

$$\Sigma \vdash G : s$$

if we can infer from the shape rules Σ that some hypergraph $G \in \mathbb{H}_{\mathcal{C}}(X)$ is shaped according to some shape name $s \in S$. For practical reasons, shape adherence must be *decidable*. However, shapes need not be *unique*: A hypergraph may be shaped according to several distinct shape names s and s' with $s \neq s'$.

We lift shapes to graphs with nesting in a compositional way, by requiring that every hypergraph in a graph is shaped. For hypergraphs at nested container positions, the shape is determined by the type of the node or edge that contains the hypergraph. For that purpose, we assume that there is a function $sh\colon \mathcal{C}_N \cup \mathcal{C}_E \cup X \to S$. For every constant node and edge $o \in \mathcal{C}_N \cup \mathcal{C}_E$, $sh(o)$ defines the *contents shape* of every container typed o; for every variable name $n \in X$, $sh(x)$ defines the *value shape* of the graphs to which x can be bound. A graph $G \in \mathbb{G}_{\mathcal{C}}(X)$ is *shaped* according to a shape name s, written $\Sigma \vdash G : s$, if

- its top hypergraph is shaped according to s, i.e. $\Sigma \vdash \hat{G} : s$, and
- the hypergraphs $G(\gamma c)$ at all nested container positions $\gamma c \in \Gamma_G$ are shaped according to the contents shape of their container objects, i.e. $\Sigma \vdash G(\gamma c) : sh(ty_{G(\gamma)}(c))$.

Edge Replacement. Let G and R be a hypergraphs with disjoint sets of nodes and edges, and let G contain an edge e with $a(R) = a(ty_G(e))$. The *replacement* of e in G by R yields the graph $G[e/R]$ as follows: Remove e from the edge set of G, add the nodes and edges of R, and identify the i-th point of R with the i-th associated node of e, for $1 \leqslant i \leqslant a(R)$.[4] In $G[e/R]$, all edges keep their associated nodes, all nodes and edges keep their types, and the points are those of G.

Context-Free Shape Rules. In papers on shape analysis for imperative programs [35, 50], shapes are usually described by logical formulas. Here we show how they can be defined as context-free hypergraph languages [13, 25].

Let $\mathcal{C}(S)$ be a *shape hypergraph* analogously to the type hypergraph $\mathcal{C}(X)$ introduced above, containing shape names S instead of variable names. We assume that S and X are disjoint. The typed and pointed hypergraphs over $\mathcal{C}(S)$ are called *syntax hypergraphs*, and denoted by $\mathbb{H}_{\mathcal{C}}(S)$. (Syntax hypergraphs without variables are just ground graphs.)

By $\langle s \rangle$ we denote the *handle hypergraph* of a shape name $s \in S$. It consists of one edge that has type s and is associated to points according to its arity. That is, $V_{\langle s \rangle} = \{n_1, \ldots, n_k\}$, $E_{\langle s \rangle} = \{e\}$, $a_{\langle s \rangle}(e) = p_{\langle s \rangle}$, and $a_{\mathcal{C}(S)}(ty_{\langle s \rangle}(e)) = ty^*_{\langle s \rangle}(a_{\langle s \rangle}(e))$.

A *context-free shape rule* $r\colon \langle s \rangle ::= R$ shall consist of the handle hypergraph $\langle s \rangle$ for some shape name s, and a syntax hypergraph R with $a(s) = a(R)$. Then r *directly derives* a syntax hypergraph G to a syntax hypergraph H, written $G \Rightarrow_r H$, if $H \cong G[e/R']$, where e is an edge in G with $ty_G(e) = s$, and the syntax hypergraph R' is isomorphic to R and disjoint to G.

Let Σ^{CF} be a finite set of context-free shape rules. We write $G \Rightarrow_{\Sigma^{\text{CF}}} H$ if $G \Rightarrow_r H$ for some rule $r \in \Sigma^{\text{CF}}$. The reflexive-transitive closure of $\Rightarrow_{\Sigma^{\text{CF}}}$ is denoted by $\Rightarrow^*_{\Sigma^{\text{CF}}}$. This allows to define context-free *shape adherence* as

$$\Sigma^{\text{CF}} \vdash G : s \;\Leftrightarrow\; \langle s \rangle \Rightarrow^*_{\Sigma^{\text{CF}}} sh(G)$$

where the *shape* $sh(G) \in \mathbb{H}_{\mathcal{C}}(S)$ of a hypergraph $G \in \mathbb{H}_{\mathcal{C}}(X)$ is obtained by replacing every variable name $x \in X$ in G by its contents shape $sh(x) \in S$.

It is decidable whether a graph satisfies some context-free shape rules Σ^{CF} or not (see [15]):

Theorem 3.1. The question "$\Sigma^{\text{CF}} \vdash G : s$?" is decidable.

Assumption 3.2. For the rest of this paper, we fix a finite set Σ of shape rules over the shape names S. In particular, we assume that S contains a distinguished subset $\tilde{S} \subset \{\tilde{s}^w \mid w \in S^*\}$ of *free shape names* such that for all $G \in \mathbb{H}_{\mathcal{C}}(X)$ with arity $a(G) = w$ satisfy $\Sigma \vdash G : \tilde{s}^w$. This allows a container c to be classified as *free* if $sh(ty_G(c)) \in \tilde{S}$.

$\mathbb{G}_\Sigma(X)$ denotes the set of Σ-*graphs*, and $\mathbb{G}_\Sigma$ denotes the set of *ground* Σ-*graphs*. Note that $\mathbb{G}_\Sigma(X)$ includes $\mathbb{G}_{\mathcal{C}}(X)$, since all containers in a graph may be free.

Example 3.2. (Context-Free Shape Rules)
Figure 6 shows the shape hypergraph used for the syntax graphs in our examples. The shape rules in Figure 8 define the shapes of the hypergraphs occurring in this paper. We use "|" to separate alternative

[4] The *identification* of two distinct nodes n and n' of the same type in a graph G keeps one of them, say n, and replaces every occurrence of the other node n' in G's point sequence, and in the association sequences of its edges, by n.

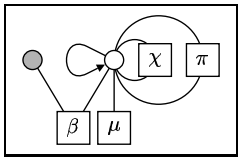


Figure 6. A shape hypergraph

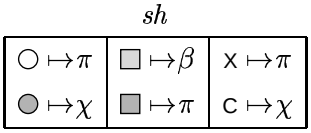


Figure 7. Shape function from $\mathcal{C}_N \cup \mathcal{C}_E \cup X$ to S.

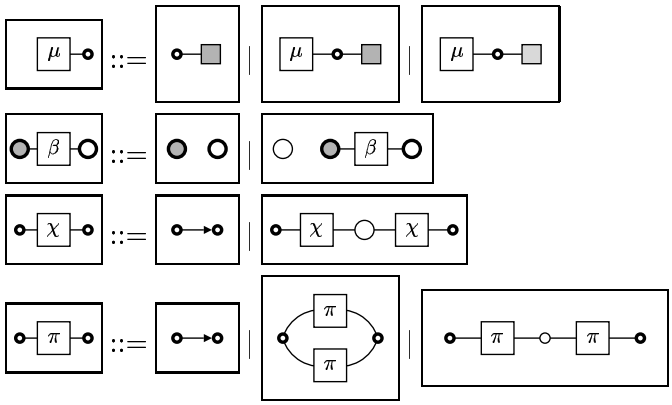


Figure 8. Shape rules for chain graphs and their contents (series-parallel graphs)

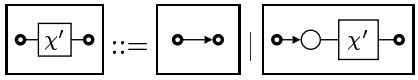
rules for the same shape name. In particular, the shape name χ defines *chain graphs*, and the shape name π defines *series-parallel graphs* as in [13, Section 2.2].

According to these rules, the hypergraphs of Example 2.1 are shaped as follows:

$$\begin{array}{llll} \Sigma \vdash G_1 & : & \mu & \qquad \Sigma \vdash G_3, G_4, G_5 \;:\; \chi \\ \Sigma \vdash G_2 & : & \beta & \qquad \Sigma \vdash G_6, G_7, G_8 \;:\; \pi \end{array}$$

The table in Figure 7 defines a shape function for the type hypergraph used in our examples. It is defined so that $\Sigma \vdash H_1 : \mu$. Note that we may also infer $\Sigma \vdash G_5 : \pi$ since every chain graph is a series-parallel graph, but then H_1 as a whole would fail to be shaped. For, the edge A in the top hypergraph G_1 has $ty_1(\mathrm{A}) = \bullet$, and $sh(\bullet) = \chi$ requires $\Sigma \vdash H_1(\mathrm{A}) : \chi$. (The graphs depicted in Figures 9, 10, 11, and 12 below are shaped as well.)

Shape rules may be *ambiguous*, i.e. generate the same graph in different ways[5]. For instance, the rules for the shape name χ are ambiguous; χ generates the same language of ground chain graphs as the shape name χ' defined by the unambiguous rules

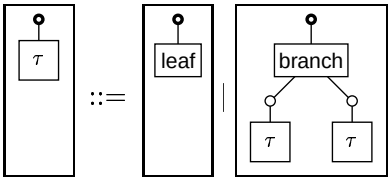


(Note that the recursive rule for χ' is obtained by applying the non-recursive rule of χ to its recursive rule so that $\Sigma \vdash G : \chi'$ implies $\Sigma \vdash G : \chi$.) For ground graphs G, χ and χ' are even equivalent. This does

[5]More precisely, a set of shape rules is *ambiguous* if some graphs in its language have more than one derivation tree, cf. [13].

not hold for chain graphs with variables, however: the unambiguous rules allow at most one variable of shape χ', which has to occur at the tail of a chain graph, whereas the ambiguous rules allow any number of variables of shape χ to occur anywhere in a chain graph. This will be exploited when we specify shape-preserving transformation rules on chain graphs in Example 4.2 below.

Generative Power of CF Shapes. Data types of functional and logic languages are tree-like. Shape rules with unary shape names suffice to define such data types: For instance, binary trees can be defined by the shape rules



(The information to be held in these trees can be stored as the contents of the leaf or branch edges.) But also data structures with sophisticated sharing, like cyclic lists, or leaf-connected trees, can be defined in a way that is not possible in imperative languages. (See [23] for a similar specification of such types in *Structured Gamma*.)

Edge replacement can only define graph shapes of bounded node degree. It is therefore impossible to define the class of complete graphs by context-free shape rules. This implies that the free shape $\tilde{s}$ can also not be defined in a context-free way. (This is no problem, however, as free shapes need not be checked.) In [15], we have extended context-free shape rules by *context-exploiting* rules, so that we can specify more general shapes without sacrificing Theorem 3.1.

For the purpose of this paper, it suffices to assume that the shape rules Σ contain context-free shapes rules, including those shown in Figure 8.

4. Transformation

Graphs are transformed by *matching* a pattern graph P, and rewriting this match with a replacement graph R. As in term rewrite rules, the pattern and replacement of graph transformation rules contain variables, and transformation is defined like term rewriting [36]: *instantiation* of the variables in P and R according to some substitution σ (of variables by graphs) generates graph instances $P\sigma$ and $R\sigma$; *embedding* of these instances into a context graph K defines transformation steps $K[P\sigma] \Rightarrow K[R\sigma]$. Both variable instantiation and context embedding are defined as nested edge replacement, a straightforward extension of edge replacement which has been described in [13], and has been used for defining context-free shapes in the previous section.

Nested Edge Replacement. Let G be a graph where the hypergraph $G(\gamma)$ at some container position $\gamma \in \Gamma_G$ contains an atomic edge e, and let R be a graph with $a(R) = a(ty_{G(\gamma)}(e))$. We assume that $\hat{R}$ and $G(\gamma)$ have disjoint nodes and edges, and that either all associated nodes of e are atomic, or so are all points of $\hat{R}$. Then the *nested replacement* of e in G by R is written $G[\gamma \cdot e/R]$; it yields the graph G' with container positions

$$\Gamma_{G'} = \Gamma_G \cup \{\gamma\bar{\gamma} \mid \bar{\gamma} \in \Gamma_R \setminus \{\varepsilon\}\}$$

that contains the hypergraphs

$$G'(\gamma') = \begin{cases} G(\gamma)[e/\hat{R}] & \text{if } \gamma' = \gamma \\ R(\bar{\gamma}) & \text{if } \gamma' = \gamma\bar{\gamma} \text{ with } \bar{\gamma} \neq \varepsilon\,, \\ G(\gamma') & \text{otherwise} \end{cases} \quad \text{for all } \gamma' \in \Gamma_{G'}$$

Context Embedding. A Σ-graph K containing a single variable e in some hypergraph $K(\gamma)$ ($\gamma \in \Gamma_K$) is an *s-context* if e has the shape $s \in S$ and is associated to atomic nodes. The *embedding* of a Σ-graph U with $\Sigma \vdash U : s$ in K is denoted as $K[U]$; it yields a graph isomorphic to $K[\gamma \cdot e/U]$.

Variable Instantiation. A function $\sigma\colon X \to \mathbb{G}_\Sigma(X)$ is a *substitution* if $\Sigma \vdash \sigma(x) : sh(x)$ and all top points of $\sigma(x)$ are atomic, for all $x \in X$.

The *instance* $P\sigma$ of a Σ-graph P according to σ is obtained by the simultaneous replacement of all x-variables in all hypergraphs of P by fresh copies of the Σ-graph $\sigma(x)$. More precisely, let $\{(\gamma_1, e_1), \ldots, (\gamma_k, e_k)\}$ be the set of all pairs such that $\gamma_i \in \Gamma_P$ and $e_i \in E_{P(\gamma_i)}$, with $ty_{P(\gamma_i)}(e_i) \in X$. Then

$$P\sigma \cong P[\gamma_1 \cdot e_1/\sigma(ty_{G(\gamma_1)}(x_1))] \cdots [\gamma_k \cdot e_k/\sigma(ty_{G(\gamma_k)}(x_k))]$$

The order of replacement is irrelevant as nested edge replacement is commutative and associative.

Example 4.1. (Substitution and Context)
The shaped graph in Figure 9 is a χ-context (since the variable name C has shape χ), and Figure 10 shows a substitution for two variables X and C.

Transformation Rules and Steps. The definition of rules and their application is similar to term rewriting [36]. Therefore, we also require the same properties as for term rewrite rules: their patterns must be more than just a variable, as other patterns apply to every graph so that transformation diverges, and their replacements must contain only variables that do already occur in their pattern; otherwise, arbitrary subgraphs may be created "out of thin air".

A *(shapely transformation) rule* $t = P/R$ consists of two Σ-graphs P and R so that $\Sigma \vdash P : s$, $\Sigma \vdash R : s$ for some $s \in S$, where the *pattern* P contains at least one constant edge, and only variable names from P occur in the *replacement* R. Then t *transforms* a graph G into another graph H, written $G \Rightarrow_t H$, if the instances of P and R according to some substitution σ can be embedded into some s-context K so that $G \cong K[P\sigma]$ and $H \cong K[R\sigma]$.

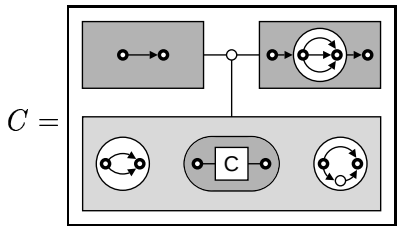


Figure 9. A context

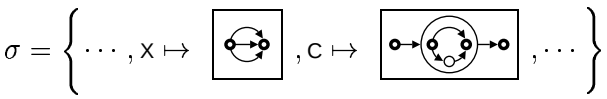


Figure 10. A substitution

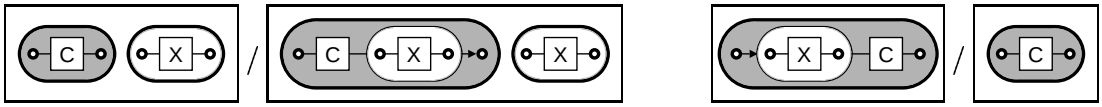


Figure 11. Two chain rules e for entering, and r for removing item graphs from chain graphs

Example 4.2. (Chain Graph Transformation)
Figure 11 shows a rule e that *enters* an item graph at the end of a chain graph, and a rule r that *removes* an item graph from the head of a chain graph. The transformation rules exploit the fact that chain graphs have the ambiguous shape χ (defined in Example 3.2) instead of the unambiguous shape χ'. The graph in the topmost container node on e's right hand side is not shaped according to χ' because the variable C with shape χ' occurs at the head of the chain. Since we may use variables of shape χ' only at the tail of chain graphs, the entering operation for nodes containing χ'-graphs could not be defined by a single rule. Instead, it had to be defined recursively, similar as a function appending items to the end of a list had to be defined in a functional language like HASKELL:

```
append x []     = [x]
append x (h:t) = h: (append x t)
```

Figure 12 shows a transformation via e, followed by a transformation via r that uses the context and substitution in Figure 9 and 10.

Variables make graph transformation quite expressive: a single step may affect subgraphs of arbitrary size: rule e *duplicates* a member node with its entire contents, since the variable name X occurs twice in its replacement graph; rule r *deletes* a member node, again with its contents, since X does not occur in its replacement graph. Let e^{-1} denote the inverse rule of e where pattern and replacement are interchanged. Then e^{-1} requires to *compare* arbitrarily large subgraphs: it applies only to a host graph, like that in the center of Figure 12, where both X-variables in its pattern bind isomorphic subgraphs. This allows to express rather complex applicability conditions. Implementations of shapely nested graph transformation may forbid such rules by requiring that their patterns are *linear*, i.e. that every variable name occurs at most once in them.

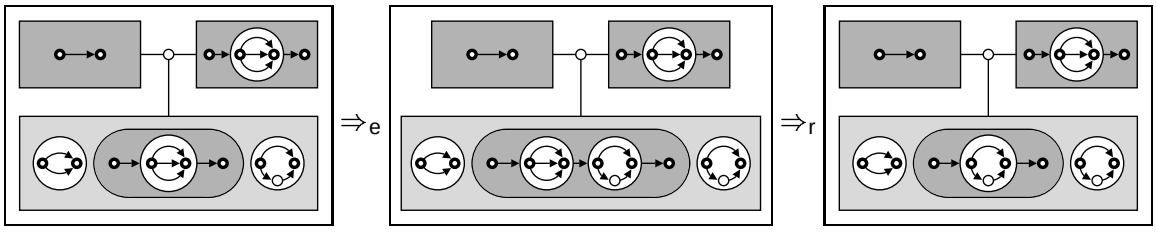


Figure 12. An entering transformation, followed by a removing transformation

Constructing Transformation Steps. The definition above does not tell how the transformed graph H can actually be constructed, given some graph G and a rule $t = P/R$. This is done as follows:

1. *Find* a subgraph $\underline{O}$ in G such that $\underline{P} \cong \underline{O} \Subset G$. Let $\gamma \in \Gamma_G$ be the deepest container position such that $\underline{P} \cong \underline{O} \Subset G.\gamma$. ($\underline{O}$ is called *skeleton occurrence* of P in G.)
2. *Bind* the variables in P by a substitution σ so that $P\sigma \cong O \Subset G.\gamma$ with $\underline{O} \Subset O$ and $O.c = G.\gamma c$ for all containers in O. (O is called *instance occurrence* of P in G.)
3. *Clip* O off G, by removing its edges, and its nodes up to the points, and attach a variable to O's points to obtain the context K.
4. *Instantiate* a fresh copy of R according to the substitution σ.
5. *Embed* the instance $R\sigma$ into K to obtain H.

Steps 1 to 5 can be done independently, for all hypergaphs contained in P in turn, starting with an occurrence of the skeleton of the top hypergraph $\hat{P}$ in some hypergraph $G(\gamma)$ of G.

In the general case, a rule may have several skeleton occurrences in a graph, and there may be several substitutions completing it to an instance occurrence. However, graph transformation is still *finitely branching*, i.e. for every graph G, there are only finitely many direct transformations $G \Rightarrow_T H$.

The paper [15] dicusses in detail the construction of transformation steps, its complexity, and some restrictions on rules that make it more efficient. Here we just note that for some Σ-graph G, every transformation $G \Rightarrow H$ and its result H is uniquely determined by a *redex* $\rho = \langle t, m, \sigma \rangle$ consisting of the applied rule $t = P/R$, an injective graph morphism m indicating the occurrence of P's skeleton in $G.\gamma$, and the substitution σ.

For graph transformation rules that are not shaped themselves, it is in general undecidable whether a transformation step preserves shapes, even if the rules do not contain variables [23, Section 4.1]. However, if graphs, contexts, and substitutions are restricted to contain shaped graphs as we do here, graph transformation preserves context-free shapes. (See [30] for the straightforward proof.)

Theorem 4.1. If $\Sigma \vdash G : s$ for some $s \in S$ and $G \Rightarrow_t H$, then $\Sigma \vdash H : s$.

Hence, shapes set up a *type discipline* that can be *statically checked*: Theorem 3.1 allows to confirm whether a transformation rule t consists of Σ-graphs or not. If this is true, and a graph G has been checked to be shaped, Theorem 4.1 guarantees that every transformation step $G \Rightarrow_t H$ yields a shaped graph H. After the step ("at runtime") type checking is not necessary.

Thus any finite set T of transformation rules constitutes a typed reduction system $\langle \mathbb{G}_\Sigma, \Rightarrow_T \rangle$ that is a powerful computational model for programming based on graph transformation.

5. Abstraction

In programming languages, *abstraction of control* means to name and parameterize compound computation tasks so that they can be used (with different actual parameters) just like elementary operations. For graph transformation, we thus need a concept for naming and parameterizing compound graph transformation tasks so that they can be used like simple transformation rules.

We call abstractions *predicates* (not functions) because graph transformation is inherently nondeterministic, so that compound transformations may fail, or deliver more than one result. We type edges by predicate names to denote predicate calls; the associated nodes of such edges designate their *actual parameters*. (Yet another way to pronounce a!) If such a node is is a container, its contents is a *graph parameter* (as in Example 6.1 below); if the contents contains predicate edges, it is a *predicate parameter* (as in Example 6.2 below). Every predicate is defined by a set of rules that may contain predicate edges in their replacement graphs by which other predicates are called.

Predicate Types. A *predicate type hypergraph* $\mathcal{C}(\mathcal{Q}, X)$ contains a type hypergraph $\mathcal{C}(X)$, plus a set $\mathcal{Q}$ of edges representing *predicate names*. We distinguish a subset $Y \subset \{x \in X \mid sh(x) \in \tilde{S}\}$ of freely shaped variables as *predicate variable names* in $\mathcal{C}(\mathcal{Q}, X)$. Edges labelled by Y are called *predicate variables*, and are placeholders for computations, whereas the variables labelled by $X \setminus Y$ are *data variables*.

Predicate Definition. Predicate edges are not part of the values of a program, but represent pending evaluation tasks during its evaluation. It does not make sense to require that predicate edges are shaped, as we do for graph values. In analogy to programming languages, graphs with predicate edges are called expressions.

Let G be a graph typed over $\mathcal{C}(\mathcal{Q}, X)$, and let $Data(G)$ denote the *data* of G, i.e. G without all predicate edges (the edges typed by $\mathcal{Q}$). G is called an *expression* if all predicate edges in G are atomic, and $Data(G)$ is a Σ-graph. Furthermore, we define $\Sigma \vdash G : s \Leftrightarrow \Sigma \vdash Data(G) : s$. We distinguish a subset $D \subset \{c \in C_G \mid sh(ty_G(c)) \in \tilde{S}\}$ of *predicate containers* in G. (We use the symbol "D" for them since the evaluation of their contents is *delayed*.) $\mathbb{E}_\Sigma(X)$ denotes the class of expressions, and $\mathbb{E}_\Sigma$ the class of *ground expressions*. Thus $\mathbb{E}_\Sigma(X) \supset \mathbb{G}_\Sigma(X)$ and $\mathbb{E}_\Sigma \supset \mathbb{G}_\Sigma$.

We lift our previous notions of contexts and substitutions to expressions: An expression K is called an *s-context* if $Data(K)$ is an s-context. A function $\sigma\colon X \to \mathbb{E}_\Sigma(X)$ is a *substitution* if $\Sigma \vdash Data(\sigma(x)) : sh(x)$ for all $x \in X$, and if $\sigma(x) \in \mathbb{G}_\Sigma(X)$ for all data variables $x \in X \setminus Y$.

An expression G is a *q-pattern* if its top hypergraph $\hat{G}$ contains exactly one predicate edge e with $ty_{G(\varepsilon)}(e) = q$, and no other hypergraph in G contains a predicate edge. The *definition* of a predicate name $q \in \mathcal{Q}$ consists of a finite, nonempty set $\mathcal{T}_q$ of graph transformation rules $t = P/R$ where P is a q-pattern, and R is an expression such that $\Sigma \vdash P : s$ and $\Sigma \vdash R : s$ for some $s \in S$. A *program* consists of a finite set $\mathcal{T} = \bigcup_{q \in \mathcal{Q}} \mathcal{T}_q$ of predicate definitions.

Lifting graph transformation to *expression evaluation* is then easy. The rewrite system $\langle \mathbb{E}_\Sigma, \Rightarrow_\mathcal{T} \rangle$ is defined as follows: For ground expressions G and H, we write $G \Rightarrow_\mathcal{T} H$ if there is an *evaluation step* via some predicate rule $t \in \mathcal{T}$, and $G \Rightarrow_\mathcal{T}^* H$ if there is an *evaluation sequence* of $n \geqslant 0$ consecutive steps. A ground expressions G is a *normal form* if no rule in $\mathcal{T}$ applies to G, and *terminal* if G is a Σ-graph. (Then $Data(G) = G$.) A program $\mathcal{T}$ is *weakly normalizing* if every ground expression G evaluates to at least one normal form, and *uniquely normalizing* if every ground expression G evaluates to at most one normal form. $\mathcal{T}$ is called *terminating* if every evaluation sequence leads to a normal form after finitely many steps.

Example 5.1. (A Predicate Type Hypergraph and a Simple Predicate)
Figure 13 shows the predicate type hypergraph used in the sequel. Predicate edges and Predicate variables are distinguished by drawing their boxes with double lines. The predicate variable Q has shape $\tilde{s}^X$. It will

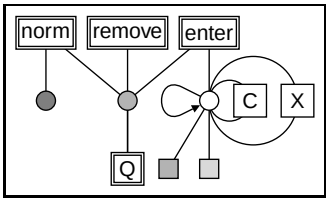


Figure 13. A predicate type hypergraph

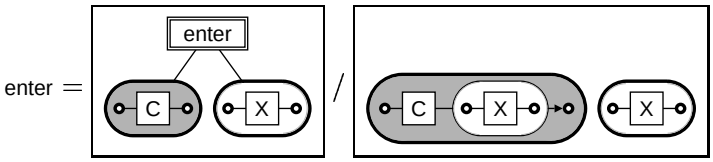


Figure 14. The predicate enter

be used to bind the contents of predicate containers in Example 6.2 further below. Predicate containers shall be drawn with double lines as well.

Figure 14 shows how the transformation rule e shown in Figure 11 of Example 4.2 can be turned into a very simple predicate enter that is defined by exactly one rule. The predicate type hypergraph makes clear that enter has the arity ●○. Since these types of nodes may contain graphs of shapes χ and π, respectively, the predicate has one chain graph and one series-parallel graph as its parameters.

Predicate Evaluation. The evaluation of a program over a ground input expression can be imagined as constructing an *evaluation tree* Δ that is a nested graph, and defined as follows: Its top hypergraph $\Delta(\varepsilon)$ is a tree of *evaluation states* that contain ground expressions, and are connected by atomic binary edges labeled with redices ρ. $\Delta(\varepsilon)$ is infinite unless the evaluation relation $\Rightarrow_{\mathcal{T}}$ is terminating. Its root n_0 represents the *initial state* and contains the ground input expression. If $\Delta(\varepsilon)$ contains an atomic binary ρ-edge from a state n to a state n', then $\Delta.n \Rightarrow_t \Delta.n'$ via some redex ρ of t. A state $n \in \Delta(\varepsilon)$ is *complete* if there is a ρ-edge from n to some state n' labelled by every redex ρ that occurs in the expression $\Delta.n$. Δ is *complete* if all its states are complete. Then, a leaf n in Δ contains an expression $\Delta.n$ that is in normal form; if $\Delta.n$ is a Σ-graph, it is a *result* of Δ, otherwise n is called a *blind alley* of the evaluation.

Practical implementations will *not* construct Δ completely, but just modify the input state during evaluation, and restore previous versions during backtracking; our simplistic assumption shall only make discussion easier.

As programs are nondeterministic in general, we define their semantics by an *evaluation function* $eval_{\mathcal{T}} \colon \mathbb{E}_\Sigma \to {\mathbb{G}_\Sigma}^\infty$ that enumerates a (possibly infinite) sequence of results one after the other, in a nondeterministic way.[6] This function initializes the evaluation tree Δ by the initial state n_0 that contains the input expression G. Then edges and successor states for evaluation steps $\Delta.n \Rightarrow_t \Delta.n'$ are added until every state is complete. Whenever $\Delta.n'$ is a Σ-graph, it is returned as a result.

Depth-First Innermost Evaluation. The evaluation function $eval_{\mathcal{T}}$ is nondeterministic in several respects:

1. Which is the *actual state* $\hat{n} \in \Delta(\varepsilon)$ where the evaluation shall continue?
2. Which is the *actual call*, i.e. the predicate edge $\hat{e}$ in the hypergraph $\Delta(\hat{n}\hat{\gamma})$ with $\hat{\gamma} \in \Gamma_{\Delta.\hat{n}}$ where the next occurrence shall be sought?
3. Which is the *actual rule* $\hat{t} \in \mathcal{T}_q$ (with $q = ty_{\Delta(\hat{n}\hat{\gamma})}(\hat{e})$) that shall be applied at this edge?

[6] A^∞ denotes the set of finite and infinite sequences over a set A.

4. Which *actual redex* $\hat{\rho} = \langle \hat{t}, \hat{m}, \hat{\sigma} \rangle$ of the actual rule $\hat{t}$ shall be used to transform $\Delta.n$?

Such a degree of nondeterminism is not only inefficient, but also confusing for programmers. Therefore we propose a general *evaluation strategy* that reduces nondeterminism:

1. States in $\Delta(\varepsilon)$ are totally ordered by age. The actual state $\hat{n}$ is the most recently inserted state that is incomplete. This strategy is known as *depth-first search* in logic programming. (*Breadth-first search*, the opposite strategy, has the advantage to determine every normal form of the input expression, but only after exploring *all* shorter evaluations, which may be very inefficient. So we accept the possibility that depth-first search may diverge due to some non-terminating rule although the input expression has a normal form.)

2. We order the predicate edges in a state n by age in the first place, and by nesting depth in the second place. This order is partial as a transformation step may introduce several predicate edges on the same nesting level. The actual call $\hat{e}$ is an innermost of the newest predicate edges in $\Delta.\hat{n}$ that occur outside of predicate containers. This strategy corresponds to *innermost evaluation* (or *eager evaluation*) in functional programming. (Again, eager evaluation does not always find all normal forms, unlike the complementary strategy of *lazy evaluation*. We prefer it because it chooses an evaluation order that we find more intuitive for programmers.)

3. Rules are ordered as they appear in the program. The actual rule $\hat{t}$ is the first rule for with a redex $\rho = \langle \hat{t}, \hat{m}, \hat{\sigma} \rangle$ that has an occurrence $\hat{m}$ containing the actual call $\hat{e}$ so that no ρ-edge issues from $\hat{n}$.

4. The actual rule $\hat{t}$ may have many redices containing $\hat{e}$. These redices may have different occurrences $\hat{m}$ that *overlap* with each other, and/or substitutions $\hat{\sigma}$ that *compete* with each other.

   Neither occurrences nor substitutions can be ordered in a canonical way. In [14] we consider conditions ensuring that the substitution $\hat{\sigma}$ is *uniquely determined* by $\hat{t}$ and $\hat{m}$. Our running example satisfies these conditions. (The rules `e` and `r` have no variable in their top hypergraphs, and at most one in each of their other hypergraphs. See [14, Theorem 1] for details.)

In the following, we assume that substitutions are uniquely determined as discussed in 4. The general evaluation strategy then leaves two sources of nondeterminism in the refined evaluation function $eval_{\mathcal{T}}$:

- Several predicate edges may be chosen as the actual call $\hat{e}$.
- The actual rule $\hat{t}$ may have several occurrences that *overlap* in the actual call.

The second source can only be avoided by careful design of shape and transformation rules. The applicability conditions proposed below allow to control the first source of nondeterminism.

6. Control

In this section we propose concepts by which programmers may control evaluation beyond the general strategy. Completion clauses allow to handle blind alleys in the evaluation, and applicability conditions order the predicates inserted by an evaluation step.

Completion Clauses. A program $\mathcal{T}$ is *sufficiently complete* [24] if every normal form of $\Rightarrow_{\mathcal{T}}$ is terminal, i.e. does not contain a predicate edge. Only if all predicate definitions are sufficiently complete, every graph has a complete evaluation. (Sufficient completeness does not suffice alone, however: evaluation must *terminate* as well.)

Sufficient completeness is a desirable property of predicate definitions; it is not so easy to achieve, however. For instance, we could define a predicate remove based on the single rule r of Example 4.2, by associating a remove-edge to the container node in its pattern. (See the two expressions in Figure 15.) This predicate would not be sufficiently complete, as its rule applies only if its actual parameter is a non-empty chain graph; otherwise, evaluation gets stuck in a blind alley. In programming languages, there are different ways to handle blind alleys:

- *Logic languages* consider them as *failure*. Then *backtracking* returns to the evaluation step where the predicate was called, and tries another evaluation starting from there.

- *Functional languages* consider such programs to be *erroneous*. Then evaluation *throws an exception* that can be caught (by *exception handlers*) in the context where the predicate has been called.

As evaluation can be nondeterministic, we cannot restrict ourselves to the functional interpretation alone. For conceptual completeness, we even allow a third way of completion: *success* is the complement of *failure*; it allows to continue evaluation. (Figure 16 shows a predicate using this kind of completion.)

Technically, success, failure and exceptions are signaled by edges labeled with built-in predicate names $+$, $-$, and $\bot_1 \ldots \bot_k$ ($k > 0$) respectively. Success and failure edges are always nullary, but exceptions may have actual parameters (to be used by the exception handlers). Completion clauses are patterns of these predefined edges. It is mandatory that every predicate definition concludes with a completion clause, following the symbol "$/\!/$". In order to make control even more explicit, we allow that a completion predicate may also occur in the replacements of other rules. (At most one!) If such a rule is applied, the completion is raised without trying to apply any further rule of that predicate.

A predicate definition may furthermore contain *exception handlers* (to catch exceptions thrown by the predicates called in its rules). They are specified by *exceptional rules* for $\bot_i$-patterns ($1 \leqslant i \leqslant k$). Exception rules start with "?", and occur just before the completion clause.

If there is no (more) actual rule $\hat{t}$ with an occurrence m containing the actual call $\hat{e}$, the completion clause in the corresponding predicate definition is executed as follows:

- For *success* ($+$), a fresh evaluation state n' is added with a $+$-edge from $\hat{n}$ to n' so that $\Delta.\, n'$ contains $\Delta.\, n$ without $\hat{e}$, and evaluation continues.

- For *failure* ($-$), backtracking determines the ancestor state $\bar{n}$ of $\hat{n}$ where $\hat{e}$ has been introduced. Evaluation continues with the next redex for the actual call of $\bar{n}$.

- For *exceptions* ($\bot_i$), interrupt handling determines the closest ancestor state $\bar{n}$ of $\hat{n}$ with an exceptional clause for $\bot_i$. This clause is evaluated, and recorded in Δ by an $\bot_i$-edge from $\bar{n}$ to a new state n' where evaluation continues.

The handling of completion clauses will be illustrated in Example 6.2 below.

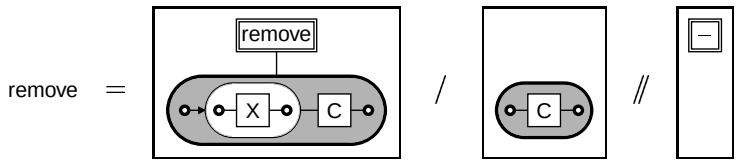


Figure 15. The predicate remove

Example 6.1. (A Predicate with Completion Clause)
In Figure 15 we define a remove predicate based on the single rule r of Example 4.2, with a completion clause that specifies that the predicate fails if its ordinary rule cannot be applied. A functional specification of remove could raise an exception $\perp_{\mathsf{ec}}$ (signaling an *empty chain*) that could then be handled by predicates calling remove. The predicate enter shown in Figure 14 would need no completion since it is sufficiently complete: It applies to every pair of nodes that contain a chain graph, and a series-parallel graph, respectively. Thus whatever completion we chose, none of them will ever occur.

Applicability Conditions. So far, the predicates inserted by an evaluation step may be evaluated in an arbitrary innermost order. However, it is often reasonable to consider some predicates as *applicability conditions* that have to be evaluated first, in order to confirm that this rule, and no other one, shall be applied, and evaluate the remaining predicates only then.

We denote a *conditional rule* for a predicate q as $P \,[\!]\, A/R$, where P is a q-pattern, while A and R are expressions, all of the same shape. A conditional rule for some predicate q is applied to a ground expression G in two stages:

1. The rule is tentatively applied, like an unconditional rule $P/A \cup R$. Then all predicate edges from A are evaluated completely.

2. If the complete evaluation of A fails, the application of the rule fails, R is removed, and another rule for q may be applied. Otherwise, applicability of the rule is confirmed, and evaluation continues. If the evaluation of some predicate from R fails, the evaluation of q fails irrevocably, without trying further rules for q.

Conditional rules work similar as the *cut* operator ! in PROLOG, by which backtracking in the evaluation of a clause

$$q_0 \text{ :- } q_1, \ldots, q_i, !, q_{i+1}, \ldots q_k$$

is cut off after q_i succeeded. Then q_1 to q_i correspond to the applicability condition in a conditional rule.

Applicability conditions suffice to make the local evaluation order in a replacement graph deterministic. For, rules can be split up so that A and R contain at most one predicate edge each.

Example 6.2. (A Control Combinator)
The control combinator norm shown in Figure 16 applies the contents of a predicate container, denoted by the unary predicate variable Q, to a chain graph container as long as this is possible. Both the predicate container and the predicate variable Q have shape $\tilde{s}^{\bigcirc}$, i.e. may contain (resp.: bound to) graphs in $\mathbb{G}_{\mathcal{C}}(X)$ with a point of type $\bigcirc$, which in turn may contain chain graphs.

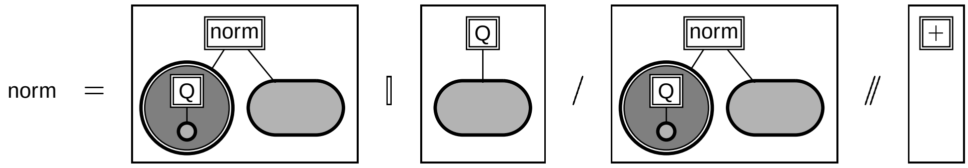


Figure 16. The control combinator norm

The regular rule calls norm recursively if the evaluation of the predicate succeeds; otherwise its completion clause makes that it resumes with success.

Figure 17 illustrates an evaluation of norm with remove as an actual predicate parameter. The first step applies the regular rule of norm, since this is the only predicate occurring outside of a predicate container. This inserts the applicability condition remove, and tentatively the recursive application of norm as well. (The tentative call is depicted with dashed lines.) Then remove is applied successfully, thus confirming the recursive application of norm to the shortened chain graph. This continues until the chain graph is empty. Then another attempt to apply remove fails so that the application of norm's regular rule fails as well, and the application of norm's completion clause makes that the evaluation terminates with success. In this case, there exists no other evaluation for the start graph of this evaluation sequence.

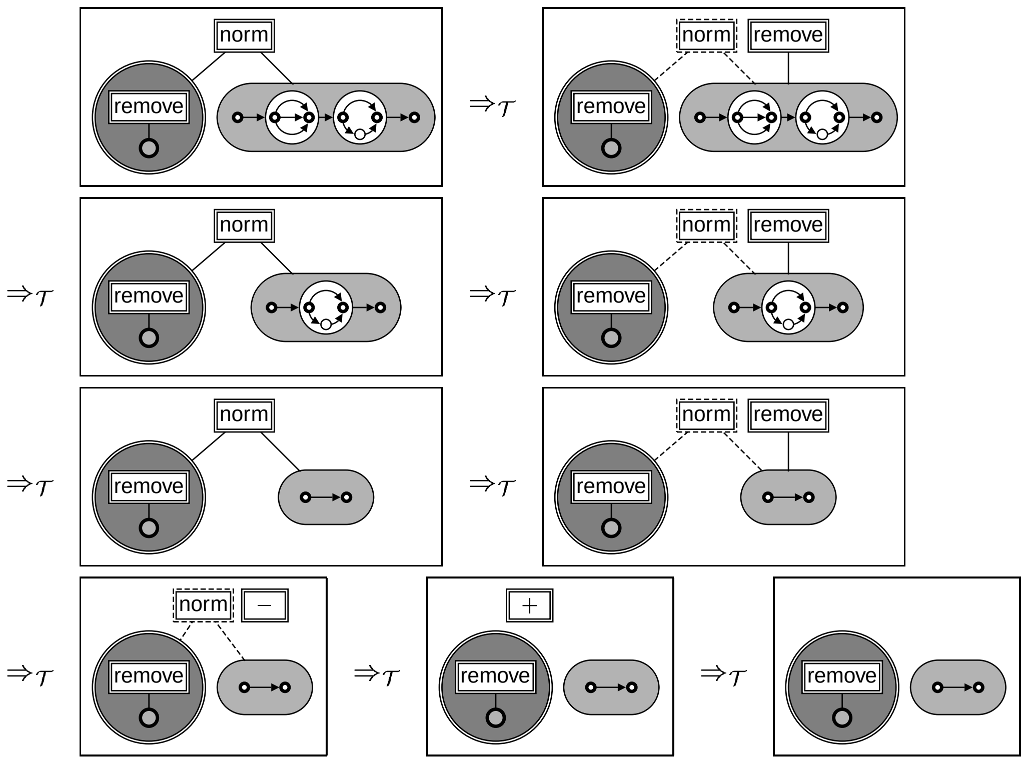


Figure 17. An evaluation of norm with remove

For the termination of norm, it is essential that Q is evaluated as the applicability condition. If the rule were unconditional, evaluation could loop in the recursive rule without ever evaluating Q.

It may be considered bad programming style to let the applicability condition have an effect on the actual parameters of norm; note however that the chain graph is only changed if remove succeeds; no actions had to be taken in order to restore the original argument if the applicability condition fails.

Other common control structures can be defined by similar combinators. Note, however, that combinators cannot be defined as easily as in functional languages, because shapes like χ are not *polymorphic*, and graph variables have a fixed arity. So the norm-combinator only applies to chain graphs and to unary predicates on chain graphs.

Failability and Nondeterminism. Completion clauses allow to determine whether a predicate may fail or not: If at least one rule of the predicate contains a fail ($-$), it may fail; otherwise, it doesn't. Unfailable predicates will be called *actions*[7]. Note that actions need not evaluate successfully in all cases: They may raise exceptions, or they may just fail to terminate. In Figure 15, remove is defined as a failable predicate; the combinator norm is an action.

Another important property of predicates (or actions) is whether they may deliver at most one result, or may yield several results. This property requires to check for confluence of rules, or more specific, for confluence of critical pairs of overlapping occurrences of rules. Even if there has been work on critical pairs for graph transformation, this is for unnested graphs and rules without variables [43]. So it is out of scope to check for determinism automatically. However, a programmer may specify whether he *intends* a predicate (or action) to be deterministic. If so, evaluation may cut off the search for further solutions after the first result has been delivered.

The predicate remove in Figure 15 is deterministic: it removes the first item graph container from a chain graph. The combinator norm is not deterministic in general because Q might be bound to a nondeterministic predicate. Predicate parameters make it impossible to check properties like determinism. However, a programmer might wish to specify that norm shall be evaluated deterministically, i.e. stop after producing the first result.

Now, coming to the end of this paper, we summarize the concepts we have introduced for predicate definitions. Taking all our considerations into account, a predicate definition might have the following structure:

$$[\mathbf{non}]\mathbf{deterministic}\,(\mathbf{predicate}|\mathbf{action})$$

$$\begin{array}{rcccccc}
q & = & P_1 & [\!] & A_1 & / & R_1 \\
 & & \vdots & & \vdots & & \vdots \\
 & | & P_m & [\!] & A_m & / & R_m \\
 & ? & E_1 & [\!] & H_1 & & \\
 & & \vdots & & \vdots & & \\
 & ? & E_n & [\!] & H_n & & \\
 & /\!/ & F & & & &
\end{array}$$

The definition of each predicate $q \in \mathcal{Q}$ specifies whether it is deterministic or nondeterministic, and wheter is is a failable predicate or an action. Its body consists of $m \geqslant 1$ conditional rules, plus $n \geqslant 0$

[7] The name is inspired by the affixgrammar language CDL2 [37].

exception clauses that specify "handlers" H_i for the exceptions E_i to be caught. And, a completion clause F specifies success or failure, or raises an exception if no other rule applies. Obviously, failure predicates are forbidden to occur in actions.

7. Conclusions

We have extended shapely nested graph transformation, a powerful model for computing with graphs that are recursively structured and shaped, by concepts for abstraction and control that shall become part of the language DIAPLAN. Predicates name and parameterize compound transformations, a global evaluation strategy (depth-first innermost) restricts nondeterminism, completion clauses cut off blind alleys of the evaluation, and applicability conditions allow programmers to control the order of predicate evaluation. The concepts are inspired by the way how term rewriting [36] is extended to functional programming languages like Haskell [42].

Related Work. We concentrate our discussion of related concepts to PROGRES [51], the (so far) most successful and comprehensive programming language based on graph transformation. PROGRES *productions* are similar to our transformation rules. They specify basic operations that are named, and may be parameterized by nodes, edges, and attribute values (also by types). PROGRES *procedures* call these productions (and other procedures), using a rich language of deterministic and nondeterministic control structures that is textual. Procedures are named and may be parameterized as well. So there is a clear separation between the graphical, rule-based specification of basic operations, and the textual, procedural programming of procedures. In contrast to that, the predicates proposed in this paper are just defined by slightly extended rules. This is more regular, and we find it more intuitive.

Another weakness of PROGRES lies in its underlying computational model, which does not support graph structuring. So productions and procedures always operate on one large graph without nesting. This is not satisfactory as program structuring has to be accompanied with data structuring in order to be effective. Our predicates profit from the nesting concept: their arguments may be graphs, and even predicates (contained in their associated nodes), not just links to atomic nodes in a global graph.

Three other pieces of related work shall briefly be mentioned. Two of them are in the area of graph transformation:

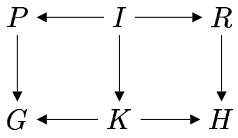
- The AGG system [22], developed at TU Berlin, is a prototyping rather than a programming system; it does not provide abstraction, and only rudimentary control structures.
- The GRACE proposal [38, 40] for a graph-centered programming language is a generic framework for structuring given graph transformation models, including the control mechanisms given in those models. Like in PROGRES, the level of rules is separate from that of abstractions, which are called *transformation units*. The control mechanisms studied for GRACE are based on path expressions [39].

Neither AGG nor GRACE do support graph structuring. This also holds for a third language, GAMMA [3], for nondeterministic computation by *multiset rewriting*. (This is badly disguised graph transformation.) The extension *Structured Gamma* defines shapes by rules analogous to context-free shape rules. The computation rules themselves are not required to be shaped, but shape adherence is determined by an

inference algorithm [23] that is (necessarily) incomplete. We are not aware of any other graph- and rule-based programming or specification language that supports recursive structured graphs and shapes, and integrates abstraction and control seamlessly so that the rule-based and graphical nature of the underlying computational model is preserved.

Moreover, DIAPLAN has a *generic user interface*. With the DIAGEN system [41], a diagram language can be specified for graphical values, mostly by graph grammars and transformation rules compatible to those discussed for specifying shapes. The diagram editors generated from the specification provide customized interactive interfaces for constructing inputs for DIAPLAN programs, and for displaying their results.

Foundations of the Computational Model. Our notion of graph transformation has been derived from *double pushout graph transformation* [17, 9, 18]. Transformation steps in this approach are defined by diagrams of the form



where the squares are pushouts in some category of graphs, e.g. hypergraphs [25]. In our work on *hierarchical graph tranformation* [16], this approach has been lifted to hypergraphs with container edges, for the case that the vertical morphisms in the diagram are injective. According to a result of [26], this does not lessen the computational power, as more general morphisms can be simulated by finite sets of quotient rules. (In both cases, double pushout graph transformation is computationally complete [53].)

The rules in [16] may have "container variables" (which may only be bound to the entire contents of a container). In [30], we have extended this concept so that variables may bind arbitrary (nested) subgraphs. As in the substitutive approach to graph transformation [46], rules are instantiated (by parallel nested edge replacement, in our case) before they are applied (by context embedding). Nested edge replacement is a simple case of hierarchical graph transformation, and context embedding corresponds to hierarchical graph transformation where the interface graph I in the diagram above is discrete, mapping the top level points of P onto those of R. Thus both rule instantiation, as application of rule instances, are special kinds of double pushout transformation. However, our approach is more than "just" an instantiation of double pushout transformation in some suitable category. So the work on *high-level replacement systems*[19] does not directly apply here.

Future Work. Some of the concepts described in this paper need further extensions in order to fit real programming needs:

- The type hypergraphs introduced in Section 3 support only *monomorphic typing*. Overloading and inheritance, like in PROGRES would be useful. The multi-level type hypergraphs are used in [21] to model type evolution; this might provide some kind of inheritance. Parameterized types, or actually parameterized shapes, like chain graphs $\chi(\alpha)$ over arbitrary item graphs α, are also desirable.

- We mentioned already that context-free shapes are rather restricted, so that we have proposed context-exploiting shapes in [15]. The *Church-Rosser shapes* of [2] seem to be still more general. Unfortunately, they do not always combine nicely with rules and rule application, because they are not preserved under instantiation. However, we could require that context-exploiting shape rules should also satisfy the Church-Rosser property. The shape rules for chain graphs and series-parallel graphs are of this kind. It seems that Church-Rosser rules characterize a kind of ambiguous shape rules that could be handled more easily in implementations.

- We want to distinguish parameter modes (*in*, *out*, and *in-out*), and distinguish whether predicates have a global effect or not (on the graph outside their parameters). With this information on *data flow*, we can characterize common programming paradigms, like functions (effect-free actions without in-out-arguments) or methods (effect-free predicates with one in-out-argument, the *receiver object*). This will make programs more transparent, and enhance their implementation.

- For a functional style of programming where predicates shall be deterministic, we need to check their definitions for *confluence*. The critical pair lemma of [44] provides confluence criteria only for hypergraph transformation without variables. However, since graphs are trees, or *terms*, of hypergraphs, there is some hope that these criteria can be combined with the critical pair lemma for term rewriting [36], which considers variables in rules.

Other concepts of DIAPLAN are still being designed:

- *Encapsulation* is important for structuring large programs as sets of modules. Modularization has been studied for graph transformation [28], but rarely in connection with nesting. Shapes could be extended to graph classes in a rather obvious way, by associating predicates as *methods* to particular shapes. For instance, the predicate remove could be made a method of chain graphs.

- Having classes like that, it is easy to integrate predefined values like numbers or strings: They are predefined classes, and may be contained in compound nodes and edges. Operations of predefined classes (like addition or string concatenation) could be invoked by corresponding predicate edges.

- *Concurrency* is still completely open. Most likely, it will be added on the level of classes, similar as in Java [1].

An implementation of a DIAPLAN interpreter, as an extension of the DIAGEN system [41], is under way. The first version will have no shapes and only rudimentary control mechanisms.

References

[1] Arnold, K., Gosling, J.: *The Java Programming Language*, Java Series, Addison-Wesley, Reading, Massachusetts, 1998, 2nd edition.

[2] Bakewell, A., Plump, D., Runciman, C.: *Specifying Pointer Structures by Graph Reduction*, Technical report, Department of Computer Science, University of York, Nov. 2002.

[3] Banâtre, J.-P., LeMétayer, D.: Gamma and the Chemical Reaction Model: Ten Years after, in: *Coordination Programming: Mechanisms, Models, and Semantics*, World Scientific, 1996, 3–41.

[4] Bardohl, R., Minas, M., Sch¨urr, A., Taentzer, G.: Application of Graph Transformation to Visual Languages, in: Engels et al. [20], chapter 3, 105–180.

[5] Barendregt, H. P.: *The Lambda Calculus—Its Syntax and Semantics*, North-Holland, Amsterdam, 1981.

[6] Berge, C.: *Graphs and Hypergraphs*, North Holland, Amsterdam, 1973.

[7] Burnett, M. M., Goldberg, A., Lewis, T. G., Eds.: *Visual Object-Oriented Programming*, Manning, 1995.

[8] Busatto, G.: *An Abstract Model of Hierarchical Graphs and Hierarchical Graph Transformation*, Dissertation, Universit¨at Paderborn, June 2002.

[9] Corradini, A., Ehrig, H., Heckel, R., L¨owe, M., Montanari, U., Rossi, F.: Algebraic Approaches to Graph Transformation Part I: Basic Concepts and Double Pushout Approach, in: Rozenberg [48], chapter 3, 163–245.

[10] Corradini, A., Ehrig, H., Montanari, U., Padberg, J.: The Category of Typed Graph Grammars and its Adjunction with Categories of Derivations, in: Cuny et al. [12], 56–74.

[11] Corradini, A., Rossi, F.: Hyperedge Replacement Jungle Rewriting for Term Rewriting Systems and Logic Programming, *Theoretical Computer Science*, **109**, 1993, 7–48.

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

[13] Drewes, F., Habel, A., Kreowski, H.-J.: Hyperedge Replacement Graph Grammars, in: Rozenberg [48], chapter 2, 95–162.

[14] Drewes, F., Hoffmann, B., Minas, M.: Constructing Shapely Nested Graph Transformations, *Proc. Int'l Workshop on Applied Graph Transformation (AGT'02)* (H.-J. Kreowski, P. Knirsch, Eds.), 2002, 107–118.

[15] Drewes, F., Hoffmann, B., Minas, M.: Context-Exploiting Shapes for Diagram Transformation, *Machine Graphics and Vision*, **12**(1), 2003, 117–132.

[16] Drewes, F., Hoffmann, B., Plump, D.: Hierarchical Graph Transformation, *Journal of Computer and System Sciences*, **64**(2), 2002, 249–283.

[17] Ehrig, H.: Introduction to the Algebraic Theory of Graph Grammars, *Proc. Graph Grammars and Their Application to Computer Science and Biology* (V. Claus, H. Ehrig, G. Rozenberg, Eds.), number 73 in Lecture Notes in Computer Science, Springer, 1979.

[18] Ehrig, H., Heckel, R., L¨owe, M., Ribeiro, L., Wagner, A., Corradini, A.: Algebraic Approaches to Graph Transformation Part II: Single Pushout Approach and Comparison to Double Pushout Approach, in: Rozenberg [48], chapter 4, 247–312.

[19] Ehrig, H., L¨owe, M.: Categorical Principles, Techniques and Results for High-Level Replacement Systems in Computer Science, *Applied Categorical Structures*, **1**(1), 1993, 21–50.

[20] Engels, G., Ehrig, H., Kreowski, H.-J., Rozenberg, G., Eds.: *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. II: Applications, Languages, and Tools*, World Scientific, Singapore, 1999.

[21] Engels, G., Heckel, R.: Graph Transformation as a Conceptual and Formal Framework for System Modelling and Evolution, *Automata, Languages, and Programming (ICALP 2000 Proc.)* (U. Montanari, J. Rolim, E. Welz, Eds.), number 1853 in Lecture Notes in Computer Science, Springer, 2000.

[22] Ermel, C., Rudolf, M., Taentzer, G.: The AGG Approach: Language and Environment, in: Engels et al. [20], chapter 14, 551–603.

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

[24] Guttag, J. V., Horning, J. J.: The Algebraic Specification of Abstract Data Types, *Acta Informatica*, **10**, 1978, 27–51.

[25] Habel, A.: *Hyperedge Replacement: Grammars and Languages*, Number 643 in Lecture Notes in Computer Science, Springer, 1992.

[26] Habel, A., M¨uller, J., Plump, D.: Double-Pushout Graph Transformation Revisited, *Mathematical Structures in Computer Science*, **11**(5), 2001, 637–688.

[27] Harary, F.: *Graph Theory*, Addison-Wesley, Reading, MA, 1969.

[28] Heckel, R., Engels, G., Ehrig, H., Taentzer, G.: Classification and Comparison of Module Concepts for Graph Transformation Systems, in: Engels et al. [20], chapter 17, 669–689.

[29] Hoffmann, B.: From Graph Transformation to Rule-Based Programming with Diagrams, *Int'l Workshop on Applications of Graph Transformations with Industrial Relevance (*AGTIVE*'99), Selected Papers* (M. Nagl, A. Sch¨urr, M. M¨unch, Eds.), number 1779 in Lecture Notes in Computer Science, Springer, 2000.

[30] Hoffmann, B.: Shapely Hierarchical Graph Transformation, *Proc. IEEE Symposia on Human-Centric Computing Languages and Environments*, IEEE Computer Press, 2001.

[31] Hoffmann, B.: Abstraction and Control for Shapely Nested Graph Transformation, *1st Int'l Conference on Graph Transformation (*ICGT*'02)* (A. Corradini, H. Ehrig, H.-J. Kreowski, G. Rozenberg, Eds.), number 2505 in Lecture Notes in Computer Science, Springer, 2002.

[32] Hoffmann, B., Minas, M.: A Generic Model for Diagram Syntax and Semantics, *ICALP Workshops 2000* (J. D. P. Rolim, et al., Eds.), number 8 in Proceedings in Informatics, Carleton Scientific, Waterloo, Ontario, Canada, 2000.

[33] Hoffmann, B., Minas, M.: Towards Rule-Based Visual Programming of Generic Visual Systems, *Proc. Workshop on Rule-Based Languages* (N. Dershowitz, C. Kirchner, Eds.), Montréal, Quebeq, Canada, September 2000.

[34] Huet, G.: Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems, *Journal of the ACM*, **27**(4), 1980, 797–821.

[35] Klarlund, N., Schwartzbach, M. I.: Graph Types, *Proc. Principles of Programming Languages*, ACM Press, New York, 1993.

[36] Klop, J. W.: Term Rewriting Systems, in: *Handbook of Logic in Computer Science* (S. Abramsky, D. M. Gabbay, T. Maibaum, Eds.), vol. 2, Oxford University Press, 1992, 1–116.

[37] Koster, C. H. A.: Using the CDL Compiler Compiler, *Compiler Construction—An Advanced Course* (F. Bauer, J. Eickel, Eds.), number 21 in Lecture Notes in Computer Science, Springer, 1976.

[38] Kreowski, H.-J., Kuske, S.: Graph Transformation Units and Modules, in: Ehrig et al. [20], chapter 15, 607–638.

[39] Kuske, S.: More about control conditions for transformation units, *Theory and Application of Graph Transformation (TAGT'98), Selected Papers* (H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg, Eds.), number 1764 in Lecture Notes in Computer Science, Springer, 2000.

[40] Kuske, S.: *Transformation Units – A Structuring Pronciple for Graph Transformation Systems*, Dissertation, Universit¨at Bremen, Fachbereich Mathematik u. Informatik, 2000.

[41] Minas, M.: Concepts and Realization of a Diagram Editor Generator Based on Hypergraph Transformation, *Science of Computer Programming*, **44**(2), 2002, 157–180.

[42] Peyton Jones, S., et al.: Haskell 98 Language and Library: The Revised Report, December 2002, Available via `http//www.haskell.org`.

[43] Plump, D.: Hypergraph Rewriting: Critical Pairs and Undecidability of Confluence, in: *Term Graph Rewriting, Theory and Practice* (M. R. Sleep, R. Plasmeijer, M. v. Eekelen, Eds.), Wiley & Sons, Chichester, 1993, 201–213.

[44] Plump, D.: *Computing by Graph Rewriting*, Habilitationsschrift, Universität Bremen, 1999.

[45] Plump, D.: Term Graph Rewriting, in: Engels et al. [20], chapter 1, 3–102.

[46] Plump, D., Habel, A.: Graph Unification and Matching, in: Cuny et al. [12], 75–89.

[47] Robinson, J. A.: *Logic: Form and Function*, Elsevier North Holland, 1979.

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

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

[50] Sagiv, M., Reps, T., Wilhelm, R.: Solving Shape-Analysis Problems in Languages with Destructive Updating, *ACM Transactions on Programming Languages and Systems*, **20**(1), 1998, 1–50.

[51] Schürr, A., Winter, A., Zündorf, A.: The PROGRES Approach: Language and Environment, in: Engels et al. [20], chapter 13, 487–550.

[52] Smolka, G., Henz, M., Würtz, J.: Object-Oriented Concurrent Constraint Programming in Oz, in: *Principles and Practice of Constraint Programming* (P. van Hentenryck, V. Saraswat, Eds.), chapter 2, The MIT Press, 1995, 29–48.

[53] Uesu, T.: A System of Graph Grammars which Generates all Recursively Enumerable Sets of Labelled Graphs, *Tsukuba J. Math.*, **2**, 1978, 11–26.

[54] Wagner, A., Gogolla, M.: Semantics of Object-Oriented Languages, in: Engels et al. [20], chapter 4, 181–211.