Abstraction and Control for Shapely Nested Graph Transformation

Berthold Hoffmann

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

Abstract. Shapely nested graph transformation is the computational model for DiaPlan, a language for programming with graphs that represent diagrams. It supports nested structuring of graphs, structural graph types (*shapes*), and graph variables. In this paper, we extend the model by two concepts that are essential for programming: *abstraction* allows compound transformations to be named and parameterized, and *control* allows the order of rule application to be specified. These concepts combine neatly with the underlying computational model, and preserve its rule-based and *graph*-ical nature.

1 Introduction

Graph transformation defines a computational model $\langle \mathcal{G}, \Rightarrow_T \rangle$ for a class $\mathcal{G}$ of graphs, by a transformation relation $\Rightarrow_T \subset \mathcal{G} \times \mathcal{G}$ that is induced by some finite set T of graph transformation rules. Various computational models of that kind are studied in [20]. If graph transformation shall be used for specification and programming, the scale of practical systems requires that the computational model $\langle \mathcal{G}, \Rightarrow_T \rangle$ is extended by concepts for structuring and typing: its data $\mathcal{G}$ should be *structured* in a nested way; its programs T should be *encapsulated* in modules; and, *abstraction mechanisms* should allow to structure the computations $\Rightarrow_T$. Furthermore, *control mechanisms* should allow to eliminate unwanted nondeterminism in the rule-based definition of $\Rightarrow_T$, and a *type discipline* should detect inconsistencies in its data and programs.

Shapely nested graph transformation [3, 12] supports nested structuring of graphs, and comes with a structural type discipline (*shapes*). Analoguously to term rewrite rules, its rules may contain *variables* so that transformation steps may move, delete or duplicate subgraphs of arbitrary size. Shapely nested graph transformation is the computational model of DiaPlan, a language for programming with graphs that is currently being designed by Frank Drewes (Umeå), Mark Minas (Erlangen), and the author [11, 13]. DiaPlan shall complement DiaGen [17], a tool for generating editors that handle the *syntax* of diagram languages, by a language and tool for programming the *semantics* of diagram languages.

In this paper, we extend shapely nested graph transformation by concepts that are essential for the design of DiaPlan: We propose an *abstraction* concept that allows to name and parameterize compound transformations in predicate definitions, and provide *control* of the evaluation order by an overall strategy (depth-first innermost evaluation), as well as by user-definable completion clauses and applicability conditions. The extension shall be seamless so as to preserve the rule-based and graphical nature of graph transformation.

The rest of the paper is structured as follows. Sections 2 to 4 recall ingredients of shapely nested graph transformation: (nested) graphs, shapes (types), and transformation. 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 our concepts to those in related languages, and outline some further research.

Acknowlegdements. I wish to thank Frank Drewes and Mark Minas for clarifying discussions about the design of DiaPlan, and the reviewers for their advice to remove technical details from sections 2–4.

2 Graphs

Our notion of graphs is tailored to programming: edges may connect an arbitrary number of nodes to model relations of any arity; nodes and edges may contain graphs in a nested fashion so that recursively structured values can be represented. We also distinguish a sequence of interface nodes at which graphs may be glued together. This extends the *nested graphs* of [3, 4, 12] where only edges can contain graphs.

Let $\mathcal{C}$ be a typed alphabet of *constant names* with an *arity function* $\mathrm{arity}\colon \mathcal{C} \to \mathcal{C}^*$.[1] The set $\mathcal{G}$ of *graphs* consists of sixtuples $G = \langle O, E, \mathrm{lab}, \mathrm{ass}, N, p\rangle$ over a finte set O of *top-level objects* with a subset $E \subset O$ of *edges* and a complementary set $V = O \setminus E$ of *nodes*; the function $\mathrm{lab}\colon O \to \mathcal{C}$ *labels* top-level objects by constant names; the function $\mathrm{ass}\colon E \to V^*$ *associates* top-level edges to sequences of top-level nodes; N is a family of (possibly empty) graphs $G.\,o \in \mathcal{G}$ contained in the objects $o \in O$ (their *direct components*); the node sequence $p \in V^*$ designates the *points* of G.[2]

We require that the point sequence p, and the association sequences $\mathrm{ass}(e)$ (for $e \in E$) do not contain repetitions, and that edges respect arity, i.e. satisfy:

$$\mathrm{lab}^*(\mathrm{ass}(e)) = \mathrm{arity}(\mathrm{lab}(e)) \text{ for all edges } e \in E$$

This corresponds to a well-known normal form of graphs that does not restrict the expressiveness of the concepts defined below. (See [10] for details.)

We call an object $o \in O$ *atomic* if $G.\,o$ is empty, or *compound* otherwise. An edge e labelled by c is called a *c-edge*. The *handle graph* $\langle c\rangle$ of a constant name c consists of an atomic c-edge that is associated to atomic points. G is called *plain*

[1] S^* is the set of *finite sequences* over some set S, including the *empty sequence* ε.
[2] Precisely, $\mathcal{G}$ has to be defined by induction over the nesting depth of objects, see [4].

if it contains no compound objects, and we define $\text{arity}(G) = \text{lab}^*(p)$. Thus $\langle c \rangle$ is plain and has the arity of the label c.

The sequences $\Omega_G = \{\varepsilon\} \cup \{o\,\omega \mid o \in O, \omega \in \Omega_{G.o}\}$ define *positions* of objects in a nested graph. They are used to select the *nested component* $G.\omega$ at some position $\omega \in \Omega_G$. The plain graph $G(\omega)$ is the nested component $G.\omega$ without its direct components.

Two graphs G and H are *isomorphic*, written $G \cong H$, if there is a bijective function $m\colon O_G \to O_H$ between their sets of top objects so that labels, associations, and points are preserved, and all corresponding components $G.o$ and $H.m(o)$ $(o \in O_G)$ are isomorphic, recursively.

Example 1 (Graphs). The figures of this paper contain graphs of some kind. We use the following conventions when drawing graphs: nodes are depicted as circles or ovals, and edges as boxes that are drawn around their label; the contents of a node or edge (if not empty) is also drawn inside. The association of an edge e to a sequence $v_1 \dots v_k$ of nodes is depicted by a bundle of lines from e to the v_i, which are called *tentacles*. Atomic binary edges are drawn like directed edges, as arrows from their first to their second associated node; their label is written aside. The "invisible label" $\sqcup$ is omitted in figures (to model unlabelled nodes and edges).

Note the difference to notions of *hierarchical graphs* that are used for system modeling [1, 6], where the objects of an underlying graph (which is plain, according to our definition) are grouped in packages that may form a hierarchy, and may have interfaces (designating some objects of a package as "public"). In hierarchical graphs, several packages may *share* an object (or a subpackage), and the associations of an edge may cross package borders. Our nesting concept forbids sharing and border-crossing edges, and is thus *compositional*: every component of a nested graph can be replaced, independently of the other ones. This is essential for programming.

3 Shapes

Usually, not every graph is a meaningful value, but only those that have a particular *shape*: chain graphs, for instance, must have a linear structure. The term *shape analysis* has come into use for inferring properties of the pointer-based data structures in imperative programs [21]. On the more abstract level of graphs, we *define* the shape of graphs (by edge replacement) so that it can be checked statically.

Edge Replacement. Let G be graph with an edge $e \in G(\omega)$. The *replacement* of e in $G.\omega$ at the position $\omega \in \Omega_G$ by some graph U with atomic points and $\text{arity}(U) = \text{arity}(lab((e))$ is defined as follows: Unite $G.\omega$ and U disjointly, redirect all tentacles at the points of U to the corresponding associated nodes of e, remove e and U's points, and insert the resulting graph for $G.\omega$.

This way of edge replacement is a straight-forward extension of hyperedge replacement [10] to nested graphs.

Shape Rules. Let S be a typed alphabet of *shape names* disjoint with $\mathcal{C}$. $(\mathcal{C} \cup S)$-labelled graphs are called *syntax graphs* if their S-edges are atomic.

Let Σ be a finite set of *shape rules* of the form $s ::= R$, consisting of a shape name s and a syntax graph R of the same arity. Σ *directly derives* a syntax graph G to a syntax graph H, written $G \Rightarrow_\Sigma H$, by replacing an s-edge in G by the graph R of some shape rule $s ::= R \in \Sigma$. The reflexive and transitive closure of $\Rightarrow_\Sigma$ is denoted by $\Rightarrow_\Sigma^*$.

For the rest of this paper, we fix a finite set Σ of shape rules over the shape names S, and use it to specify the shape of graphs.

Example 2 (Shape Rules for Chain Graphs). The rules in Figure 1 define the shape of *chain graphs*. (We use "|" to seperate alternative rules for the shape name γ.) Two atomic points designate the begin and end of a chain, and every node in between contains an item graph. In the examples of this paper, the item graphs (with shape named ι) are series-parallel graphs as defined in [2, Section 2.2]. Figures 4 and 7 below contain chain and item graphs that are shaped according to these rules.

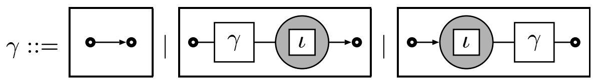

Fig. 1. Shape rules for chain graphs

The rules for γ are *ambiguous*: If we removed the rightmost shape rule, γ would still generate the same set of chain graphs. However, with this rule, we may define transformation rules in a more general way. (See Example 4 below.)

Σ-graphs. For rules, we need special edges that denote variable parts in a graph. For that purpose, we consider a typed alphabet X of *variable names* disjoint with $\mathcal{C}$ and S. We assume that every variable name is associated with a *shape name* $\mathrm{shape}(x) \in S$.

Let G be a $(\mathcal{C} \cup X)$-labelled graph where all *variables* (the X-edges) are atomic. G is called a *Σ-graph* if $\langle s \rangle \Rightarrow_\Sigma^* \mathrm{shape}(G)$ for some shape name $s \in S$, where $\mathrm{shape}(G)$ is the syntax graph obtained by relabelling every variable name in G by its shape name. Then we write $\Sigma \vdash G : s$. (Σ-graphs may have several shapes, because with a "chain" rule like $s ::= \langle s' \rangle$, $\Sigma \vdash G : s'$ implies $\Sigma \vdash G : s$.)

$\mathcal{G}_\Sigma(X)$ denotes the set of *Σ-graphs*, and $\mathcal{G}_\Sigma$ denotes the set of *constant Σ-graphs*, which contain no variables.

It is decidable whether a graph satisfies some shape rules Σ or not (see [3]):

Theorem 1. *The question "$\Sigma \vdash G : s$?" is decidable.*

Data types of functional and logical languages are tree-like. They can be defined by shape rules with unary shape names only. 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 [8] for a similar specification of such types in *Structured Gamma*.)

Edge replacement can only define graph shapes of bounded node degree. However, this restriction can be overcome without sacrificing decidability if we allow rules similar to the embedding rules used in the DiaGen system [17]. Then also shapes like that of all control flow graphs can be defined.

4 Transformation

Graphs are transformed by *matching* a pattern P, and rewriting this match with a replacement R. In the case of nested graph transformation [12], rules consist of Σ-graphs P and R, and transformation is defined like term rewriting [14]: P is embedded into some context C, after substituting its variables by appropriate graphs; transformation yields a graph where R is embedded into the same context C, after instantiating its variables with the same substitution. (The variable concept is inspired by [19].) Here we just consider "shapely" nested graph transformation. The "unshaped" case just forgets about typing. (See [12] for details.)

Context Embedding. A Σ-graph C containing a single variable e in some plain component $C(\omega)$ ($\omega \in \Omega_C$) 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 C is denoted as $C[U]$ and defined as follows: Unite $C.\omega$ and U disjointly, redirect all tentacles at the associated nodes $\mathrm{ass}(e)$ to the corresponding points of U, remove e with its associated nodes, and assign the result to $P.\omega$. (Context embedding preserves the points of the inserted Σ-graph, and removes the associations of the replaced variable, while edge replacement does it the other way round. Otherwise, both operations are equal.)

Example 3 (Substitution and Context). Figure 2 shows a γ-context (if we assume that the variable name H has shape γ), and a substitution for two variables X and C that are nullary and binary, respectively.

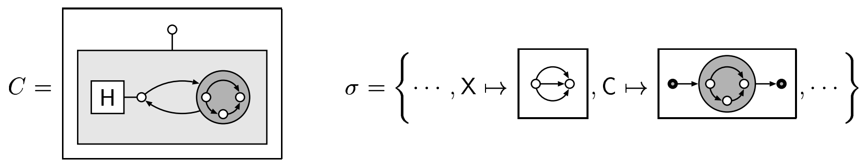

Fig. 2. A context and two substitution pairs

Variable Instantiation. A function $\sigma \colon X \to \mathcal{G}_\Sigma(X)$ is a *substitution* if $\Sigma \vdash \sigma(x) : \mathrm{shape}(x)$ for all $x \in X$.

The *instantiation* of a Σ-graph P according to σ is obtained by the simultaneous replacement of all x-variables in P by the Σ-graph $\sigma(x)$. The resulting *instance* is denoted by $P\sigma$. (The order of replacement is irrelevant as edge replacement is commutative and associative.)

Transformation Rules and Steps. The definition of rules and their application is similar as in term rewriting [14]. Therefore, we also require the same properties as for term rewrite rules: their patterns must not be variables, as such rules apply to every graph so that transformation diverges, and their replacements must not contain variables that do not occur in their pattern, since then arbitrary subgraphs may be created "out of thin air".

A *(transformation) rule* $t = P/R$ consists of two Σ-graphs P and R so that $\Sigma \vdash P : s, R : s$ for some $s \in S$, where the *pattern* P is not a variable handle, 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 a substitution σ can be embedded into some s-context C so that $G \cong C[P\sigma]$ and $H \cong C[R\sigma]$.

Graph transformation preserves shapes. (See [12] for the straightforward proof.)

Theorem 2. *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 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 2 guarantees that every transformation step $G \Rightarrow_t H$ yields a shaped graph H. After the step ("at runtime") type checking is not necessary.

Example 4 (Chain Graph Transformation). Figure 3 shows a rule e that *enters* an item graph at the end of a chain graph, and a rule r that *removes* the first item graph from a chain graph. The variable names C and X have $\mathrm{shape}(\mathsf{C}) = \gamma$ and $\mathrm{shape}(\mathsf{X}) = \iota$, respectively.

Note that the second recursive rule in Figure 1 is needed to derive the shape of the chain graph in r's pattern. Without that ambiguous shape rule, r had to be defined by recursive traversal of the chain graph.

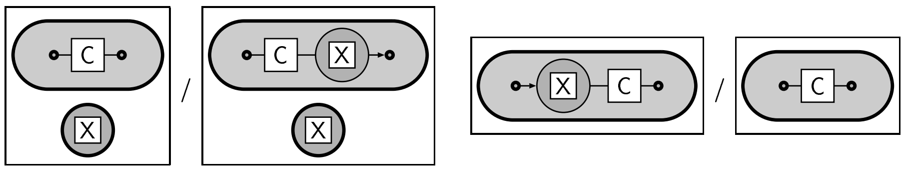

Fig. 3. Chain rules for entering and removing item graphs

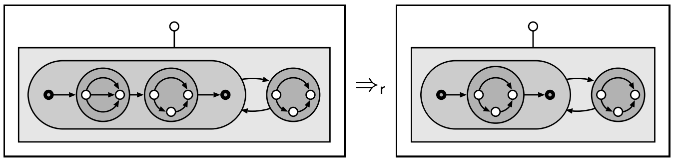

Fig. 4. A removing transformation

Figure 4 shows a transformation via r, using the context and substitution in Figure 2.

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 is 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 G, where both X-variables in its pattern match isomorphic subgraphs. This allows to express rather complex applicability conditions. Implementations of shapely nested graph transformation may forbid such rules by requiring that their pattern is *linear*, i.e. that every variable name occurs at most once.

Constructing Transformation Steps. In [3] we have discussed how transformation steps can be constructed. 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 rule $t = P/R$ used, an occurrence morphism m indicating the place where the skeleton of its pattern P (i.e. without its variables) occurs in G, and the matching substitution σ.

5 Abstraction

In programming languages, *abstraction* means to name and parameterize compound computation tasks so that they can afterwards be called (with different arguments) just like elementary computations. For graph transformation, we thus need a concept for naming and parameterizing sequences of graph transformation steps. We extend the set $\mathcal{C}$ of constant names (denoting data) by names that denote abstractions. Abstractions are called *predicates* (not functions) because their evaluation may fail, and may be nondeterministic, i.e. yield more than one result. Every predicate is defined by a set of rules that may contain predicates in their replacement graphs by which other abstractions are called.

Predicate Definition. We consider a typed alphabet Q of *predicate names* that is disjoint to $\mathcal{C}$, $\mathcal{S}$, and X. Let G be a graph labelled by $Q \cup \mathcal{C} \cup X$. $\overline{G}$ denotes the *data* of G, i.e. G without all predicate edges (the edges labelled by Q). G is an *expression* if $\overline{G}$ is a Σ-graph, and if its predicate edges are atomic. An expression G has the *shape* of $\overline{G}$. $\mathcal{E}_\Sigma(X)$ denotes the set of expressions, and $\mathcal{E}_\Sigma$ the set of *constant expressions* (without variables). Thus $\mathcal{E}_\Sigma(X) \supset \mathcal{G}_\Sigma(X)$ and $\mathcal{E}_\Sigma \supset \mathcal{G}_\Sigma$. We extend substitutions so that they map variable names to expressions of the same shape, and extend contexts and instances to be expressions.

An expression G is a *p-pattern* if it contains exactly one predicate edge e, which is labeled by the predicate name p, and occurs on its top-level $G(\varepsilon)$. The *definition* of a predicate name $p \in Q$ consists of a finite, nonempty set T_p of rules $t = P/R$ such that P is a p-pattern, and R is an expression. A *program* consists of a set $\mathcal{T} = \bigcup_{p \in Q} T_p$ of predicate definitions. (Examples of predicate definitions are given in the next section, after introducing concepts for control.)

For constant expressions G and H, we write $G \Rightarrow_\mathcal{T} H$ if there is an *evaluation step* via a transformation $t \in \mathcal{T}$, and $G \Rightarrow^*_\mathcal{T} H$ if there is an *evaluation sequence* of $n \geqslant 0$ consecutive steps. A constant expressions G is a *normal form* if no rule in $\mathcal{T}$ applies to G, and *terminal* if G is a Σ-graph. A program $\mathcal{T}$ is called *terminating* if every evaluation sequence leads to a normal form after finitely many steps. $\mathcal{T}$ is *uniquely normalizing* if every graph G evaluates to at most one normal form.

The nodes associated to a predicate edge designate its *parameters*. If such a node is compound, its contents is a *graph parameter* (as in Example 5 below); if the contents contains predicate edges, it is a *predicate parameter* (as in Example 6 below). Predicates may also match and modify the "local context" $G.\omega$ where a predicate edge e occurs if a part of one of their patterns is not contained in their parameter nodes. (However, such rules do not occur in this paper.)

Predicate Evaluation. The evaluation of a program over a constant input expression can be imagined as constructing an *evaluation tree* Δ that is defined as follows: Its top level $\Delta(\varepsilon)$ is a tree of *evaluation states* that contain constant expressions, and are connected by atomic binary edges labelled with redices ρ. Δ may be infinite unless the evaluation relation $\Rightarrow_\mathcal{T}$ is terminating. Its root v_0 represents the *initial state* and contains the constant input expression. If $\Delta(\varepsilon)$ contains an atomic binary ρ-edge from a state v to a state v', then $\Delta.v \Rightarrow_t \Delta.v'$ via some redex ρ of t. A state $v \in \Delta(\varepsilon)$ is *complete* if there is a ρ-edge from v to some state v' for every redex ρ in the expression $\Delta.v$. Δ is *complete* if all its states are complete. Then, a leaf v in Δ contains an expression $\Delta.v$ that is in normal form; if $\Delta.v$ is a Σ-graph, it is a *result* of Δ, otherwise v is called a *blind alley* of the evaluation. (Practical implementations will *not* construct Δ, but just update the input expression G; this simplistic assumption shall only make discussion easier.)

As programs are nondeterministic in general, we define their semantics by an *evaluation function* $eval_\mathcal{T}\colon \mathcal{E}_\Sigma \to \mathcal{G}^*_\Sigma$ that enumerates a sequence of results one after the other, in a nondeterministic way. This function initializes the evaluation tree Δ by the initial state v_0 that contains the input expression G. Then edges and successor states for evaluation steps $\Delta . v \Rightarrow_t \Delta . v'$ are added until every state is complete. Whenever $\Delta . v'$ is a Σ-graph, it is returned as a result.

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

1. Which is the *actual state* $\hat{v} \in \Delta(\varepsilon)$ where the evaluation shall be continued?
2. Which is the *actual call*, i.e. the predicate edge $\hat{e}$ in $\Omega_{\Delta . \hat{v}}$ where the next occurrence shall be sought?
3. Which is the *actual rule* $\hat{t} \in \mathcal{T}$ that shall be applied at this edge?
4. Which *actual redex* $\hat{\rho} = \langle \hat{t}, \hat{m}, \widehat{\sigma} \rangle$ of $\hat{t}$ shall be used to transform $\Delta . v$?

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 $\widehat{v}$ 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 is very inefficient. So we accept the possibility that depth-first search may diverge due to a non-terminating rule although the input expression has a normal form.)
2. We order the predicate edges in a state v 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 $\Delta . \widehat{v}$. 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 is more intuitive for programmers.)
3. Rules are ordered as they appear in the program. The actual rule $\hat{t}$ is the first rule for which a redex $\rho = \langle \hat{t}, m, \sigma \rangle$ has an occurrence m containing the actual call $\hat{e}$ so that no ρ-edge issues from $\hat{v}$.
4. The actual rule $\hat{t}$ may have many redices containing $\hat{e}$. These redexes may have different occurrences m that *overlap* with each other, and/or matching substitutions σ that *compete* with each other.
   Neither occurrences nor substitutions can be ordered in a canonical way. In [3] we consider conditions ensuring that the matching substitution $\widehat{\sigma}$ is *uniquely determined* by $\hat{t}$ and m. Our running example satisfies these conditions. (The rules e and r have no variable on top level, and at most one in each of their nested components. See [3, 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 *overlapping* occurrences that contain 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* [9] if every normal form of $\Rightarrow_{\mathcal{T}}$ is terminal, i.e. does not contain a Q-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, by associating a remove-edge to the compound chain node in its pattern. (See the two expressions in Figure 5.) This predicate would not be complete, as its rule applies only if its 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:

- *Logical languages* consider them as *failure.* Then *backtracking* returns to the evaluation step where p was called, and tries another evaluation step 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 6 shows a predicate using this kind of completion.)

Technically, success, failure and exceptions are signaled by edges labelled 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 parameters (to be used by the exception handlers). Completion clauses are patterns of these predefined edges. They are added at the end of every predicate definition, following the symbol "//".

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 v' is added with a +-edge from $\widehat{v}$ to v' so that $\Delta.\,v'$ contains $\Delta.\,v$ without $\hat{e}$, and evaluation continues.
- For *failure* (−), backtracking determines the ancestor state $\bar{v}$ of $\widehat{v}$ where $\hat{e}$ has been introduced. Evaluation continues with the next redex for the actual call of $\bar{v}$.
- For *exceptions* ($\bot_i$), interrupt handling determines the closest ancestor state $\bar{v}$ of $\widehat{v}$ with an exceptional clause for $\bot_i$. This clause is evaluated, and recorded in Δ by an $\bot_i$-edge from $\bar{v}$ to a new state v' where evaluation continues.

Example 5 (A Predicate with Completion Clause). In Figure 5 we define a remove predicate based on the single rule r of Example 4, 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 $\bot_{\mathsf{ec}}$ (signalling an *empty chain*) that could then be handled by predicates calling remove.

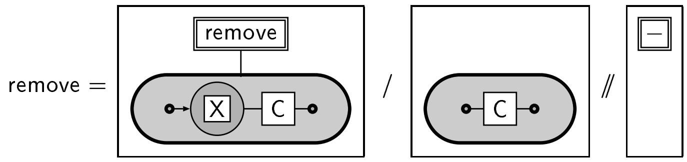

Fig. 5. The predicate remove

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 make sure that this rule shall be applied, and evaluate the remaining predicates only then.

We thus distinguish a subset A of the predicate edges in R as *applicability predicates*, and give their evaluation priority over the rest. Applicability predicates are drawn as predicate edges with a dashed outer border. This simple concept suffices to make the local evaluation order in a replacement graph deterministic. (Rules can be split up so that A and $\Omega_R \setminus A$ contain at most one predicate edge each.)

Predicate Variables. As a last extension of rules, we consider "higher-order" variables that may be bound to computations instead of data. These variables have to be distinguished only because their substitutions are different.

Let $Y \subset X$ denote a subset of *predicate variable names*, which we draw as boxes with double borders. In a substitution σ, a predicate variable $y \in Y$ may be mapped onto an expression $\sigma(y)$ that does not contain constant names, and satisfies $\text{arity}(\sigma(y)) = \text{arity}(y)$. Predicate variables may be used to program combinators in a functional style.

Example 6 (A Control Combinator). Figure 6 shows a control combinator "!" that normalizes a chain graph denoted by C according to some unary predicate denoted by the predicate variable Q. It evaluates Q as an applicability predicate, and calls itself recursively as long as this succeeds.

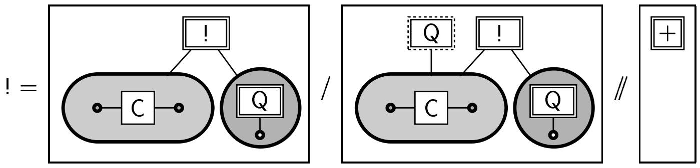

Fig. 6. The control combinator !

For the termination of !, it is crucial that Q is an applicability predicate: otherwise, the evaluation of ! could loop in its recursive rule without ever evaluating Q.

Figure 7 shows an evaluation of ! with remove as a predicate parameter. The evaluation is deterministic: it empties the chain graph given as the first parameter. (The combinator ! is not deterministic in general because it might be applied to a nondeterministic predicate Q.)

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 γ are not *polymorphic*, and graph variables have a fixed arity. So the !-combinator only applies to chain graphs and to unary chain predicates.

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

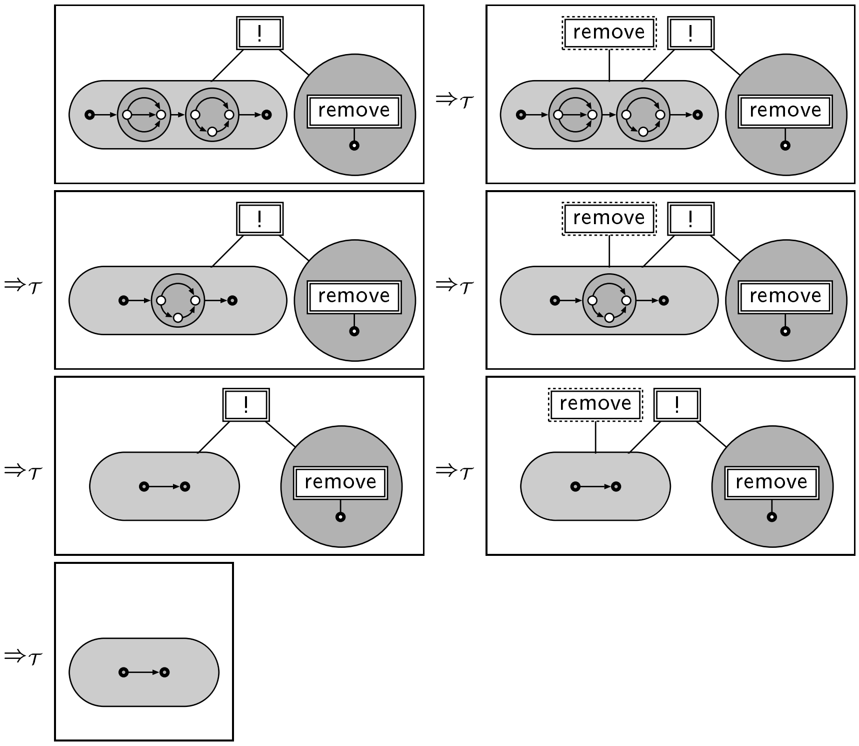

Fig. 7. An evaluation of ! with remove

clauses cut off blind alleys of the evaluation, and applicability conditions control the order of predicate evaluation. The concepts are inspired by the way how term rewriting [14] is extended to functional programming languages.

Related Work. We concentrate our discussion of related concepts to Progres [22], the (so far) most successful and comprehensive programming language based on graph transformation. Progres *productions* are simlar 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 plain graph. 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 parameters may be graphs, and even by predicate expressions (contained in their associated nodes), not just "pointers" in a global graph.

Two other pieces of related work shall briefly be mentioned. Agg [7] is a prototyping rather than a programming system; it does not provide abstraction, and only rudimentary control structures. Grace [15, 16], a generic framework for structuring given graph transformation models, also separates rules from abstractions (called *transformation units*).

We are not aware of any other graph- and rule-based programming or specification language that supports recursive structuring and typing of graphs, and integrates abstraction and control seamlessly in a rule-based fashion.

Future Work. Some concepts of DiaPlan are still open, e.g. encapsulation and concurrency, and the concepts mentioned in this paper need further thought as well.

We want to specify whether predicates may fail or not, whether they are uniquely normalizing, or nondeterministic, and whether they have an *effect* on the local context where they are applied, or not (or only if they succeed). We want to distinguish parameter *modes* (*in*, *out*, and *inout*) in order to specify *data flow*. Then we can characterize common programming paradigms, like functions (non-failing effect-free predicates without inout-parameters) or methods (effect-free predicates with one inout-parameter, the *receiver* object). This will make programs more transparent, and enhance their implementation.

If a functional style of programming shall be supported, we need to check programs for *unique normalization*. There is some hope that results concerning confluence and termination of term rewriting [14] and plain graph transformation [18] can be combined for that purpose.

And, last but not least, DiaPlan must be *implemented*.

References

1. G. Busatto. *An Abstract Model of Hierarchical Graphs and Hierarchical Graph Transformation.* Dissertation, Universität Paderborn, June 2002.
2. F. Drewes, A. Habel, and H.-J. Kreowski. Hyperedge replacement graph grammars. In Rozenberg [20], chapter 2, pages 95–162.
3. F. Drewes, B. Hoffmann, and M. Minas. Constructing shapely nested graph transformations. In H.-J. Kreowski and P. Knirsch, editors, *Proc. Int'l Workshop on Applied Graph Transformation (AGT'02)*, 2002. 107–118.
4. F. Drewes, B. Hoffmann, and D. Plump. Hierarchical graph transformation. *Journal of Computer and System Sciences*, 64(2):249–283, 2002.

5. G. Engels, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. II: Applications, Languages, and Tools.* World Scientific, Singapore, 1999.
6. 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.
7. C. Ermel, M. Rudolf, and G. Taentzer. The AGG approach: Language and environment. In Engels et al. [5], chapter 14, pages 551–603.
8. P. Fradet and D. Le Métayer. Structured Gamma. *Science of Computer Programming*, 31(2/3):263–289, 1998.
9. J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–51, 1978.
10. A. Habel. *Hyperedge Replacement: Grammars and Languages.* Number 643 in Lecture Notes in Computer Science. Springer, 1992.
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. Shapely hierarchical graph transformation. In *Proc. IEEE Symposia on Human-Centric Computing Languages and Environments*, pages 30–37. IEEE Computer Press, 2001.
13. B. Hoffmann and M. Minas. Towards rule-based visual programming of generic visual systems. In N. Dershowitz and C. Kirchner, editors, *Proc. Workshop on Rule-Based Languages*, Montréal, Quebeq, Canada, Sept. 2000.
14. J. W. Klop. Term rewriting systems. In S. Abramsky, D. M. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 1–116. Oxford University Press, 1992.
15. H.-J. Kreowski and S. Kuske. Graph transformation units and modules. In Ehrig et al. [5], chapter 15, pages 607–638.
16. S. Kuske. *Transformation Units – A Structuring Pronciple for Graph Transformation Systems.* Dissertation, Universität Bremen, Fachbereich Mathematik u. Informatik, 2000.
17. M. Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming*, 44(2):157–180, 2002.
18. D. Plump. *Computing by Graph Rewriting.* Habilitationsschrift, Universität Bremen, 1999.
19. D. Plump and A. Habel. Graph unification and matching. In 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, pages 75–89. Springer, 1996.
20. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations.* World Scientific, Singapore, 1997.
21. 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.
22. A. Schürr, A. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In Engels et al. [5], chapter 13, pages 487–550.