URL: http://www.elsevier.nl/locate/entcs/volume2.html 10 pages

A Compositional Approach to Structuring and Refinement of Typed Graph Grammars [1]

Andrea Corradini [a] Reiko Heckel [b]

[a] *Dipartimento di Informatica, Università di Pisa, I-56125 Pisa, Italy*

[b] *Fachbereich 13 Informatik, Technische Universität Berlin, D-10587 Berlin, Germany*

Abstract

Based on a categorical semantics that has been developed for typed graph grammars we uses colimits (pushouts) to model composition and (reverse) graph grammar morphisms to describe refinements of typed graph grammars. Composition of graph grammars w.r.t. common subgrammars is shown to be compatible with the semantics, i.e. the semantics of the composed grammar is obtained as the composition of the semantics of the component grammars. Moreover, the structure of a composed grammar is preserved during a refinement step in the sense that compatible refinements of the components induce a refinement of the composition. The concepts and results are illustrated by an example.

1 Introduction

This contribution addresses the structuring and refinement of typed graph grammars defined according to the algebraic double pushout approach [5]. *Typed* graph grammars are introduced in [2] for the double pushout approach (cf. [6] for a corresponding notion in the single-pushout setting). They generalize the concept of labeling graphs by providing different type sets for nodes and edges and imposing a graphical structure on it. In [1] typed graph grammars have been given a categorical semantics that generalizes similar results for P/T nets in [8]: Such a semantics is strongly based on the typing mechanism, because non-trivial grammar morphisms could be defined by exploiting the "type graphs" of the grammars. In particular three categories have been introduced: **GraGra** having typed graph grammars as objects, and grammar morphisms as arrows; **GraTS** with (typed) graph transition systems as objects and **GraCat** having small categories of (typed) graph derivations as objects. The main result of [1] shows that there are left adjoint functors TS : **GraGra** $\rightarrow$ **GraTS** and DS : **GraGra** $\rightarrow$ **GraCat** to the forgetful

[1] Research partially supported by the COMPUGRAPH Basic Research Esprit Working Group n. 7183

functors U : **GraTS** $\rightarrow$ **GraGra** and V : **GraCat** $\rightarrow$ **GraGra**, respectively. In particular, the functor DS associates with each typed grammar **G** its *derivation system* $DS(\mathbf{G})$, which is a category having graphs as objects and graph derivations as arrows, and can be considered reasonably as an "operational" semantics of the grammar. Indeed, in the rest of this contribution, by "semantics" of a grammar we shall mean its derivation system.

Graph grammars have been shown to be adequate for the specification of software systems for example in [4], and thanks to their typing mechanism, typed grammars are even more expressive in this application field. However, since real systems tend to be very large, suitable techniques for structuring specifications are needed. On the other hand, large specifications are usually not written from scratch, but they require a number of development steps. In a top-down development these are refinement steps, where an abstract specification of the system is replaced by a more specific one.

Any reasonable proposal of structuring mechanisms for (typed) graph grammars, however, should be compatible with their semantics. Operations that (syntactically) combine small graph grammars to build larger ones should have semantical counterparts doing a corresponding construction for their derivation systems. This is usually called a *compositional* semantics, meaning the ability to construct the semantics of some composed specification out of the semantics of its components. On the other hand a refinement step should preserve the structure of a specification, that is compatibility of structuring and refinement is required.

In this contribution colimits, in particular pushouts, are used as the composition mechanism of typed graph grammars, in the spirit of [3]. Moreover, refinement of grammars is modeled by grammar morphisms in the reverse direction. A very detailed example of a graph grammar specifying some operations on a list of lists is used to motivate the adequacy of these notions. The main result shows the compatibility of structuring and semantics, as well as of structuring and refinement, proving in this way the compositionality of our approach.

2 Technical Background

In this section we introduce the basic notions of typed graph grammar and grammar morphism, including their semantics, and show the existence of colimits in the corresponding category $\mathbf{GraGra}^+$. Note that, compared to the category of typed grammars **GraGra** introduced in [1], our simplified category $\mathbf{GraGra}^+$ is obtained by restricting the allowed morphisms and by ignoring the start graphs.

Let **Graph** be the category of (unlabeled) graphs and total graph morphisms and $\mathbf{Graph}^{\mathbf{P}}$ the category of graphs and partial graph morphisms, where a partial graph morphism $s : L \rightarrow R$ is a span (i.e., a pair of coinitial morphisms) $s = L \xleftarrow{l_s} K \xrightarrow{r_s} R$ in **Graph** such that l_s is an inclusion. Composition of two partial graph morphisms $s_i = L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i$, $i = 1, 2$

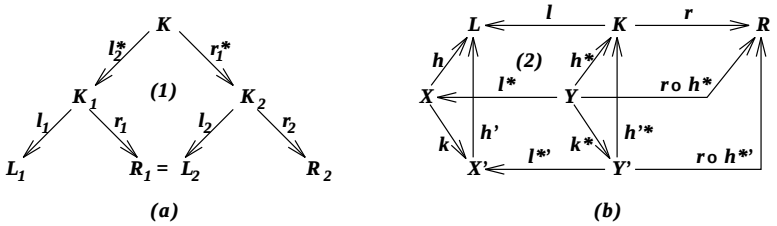

Fig. 1. Definition of the functor $\langle s \rangle : (\mathbf{Graph} \downarrow \mathbf{L}) \to (\mathbf{Graph} \downarrow \mathbf{R})$.

with $L_2 = R_1$ is defined as $s_1; s_2 = L_1 \xleftarrow{l_1 \circ l_2^*} K \xrightarrow{r_2 \circ r_1^*} R_2$ where $K \subseteq K_1$ is the inverse image of K_2 under r_1, l_2^* is the corresponding inclusion, and r_1^* is the restriction of r_1 to K. It is well-defined since the inverse image squares (1) in Figure 1 (a) (which is a special pullback) preserves inclusions. $\mathbf{Graph}^{\mathbf{P}}$ has extensively been studied in the algebraic single-pushout approach [7] and it is known to be complete and co-complete.

For $TG \in |\mathbf{Graph}|$ we denote by TG-$\mathbf{Graph}$ the category of TG-typed graphs, i.e., the comma category $(\mathbf{Graph} \downarrow \mathbf{TG})$, where objects are typed graphs $\langle H, h \rangle$ with $h : H \to TG$, and a typed graph morphisms $f : \langle H, h \rangle \to \langle H', h' \rangle$ is a graph morphism $f : H \to H'$ such that $h' \circ f = h$. Accordingly TG-$\mathbf{Graph}^{\mathbf{P}}$ is the category of typed graphs and typed partial graph morphisms, i.e. pairs of coinitial arrows $L \xleftarrow{l} K \xrightarrow{r} R$ in TG-$\mathbf{Graph}$ where l is an inclusion.

Each inclusion $l : K \to L$ of graphs induces a functor $l^< : (\mathbf{Graph} \downarrow \mathbf{L}) \to (\mathbf{Graph} \downarrow \mathbf{K})$ defined on objects as $l^<(\langle X, h \rangle) = \langle Y, h^* \rangle$, where diagram (2) in Figure 1(b) is an inverse image square, and on arrows as $l^<(k : \langle X, h \rangle \to \langle X', h' \rangle) = k^*$, where since the square $LKY'X'$ of Figure 1(b) is a pullback, k^* is uniquely determined because $l \circ h^* = h' \circ k \circ l^*$, and it satisfies $h'^* \circ k^* = h^*$ and $k \circ l^* = l^{*'} \circ k^*$. On the other hand, each total graph morphism $r : K \to R$ induces a functor $r^> : (\mathbf{Graph} \downarrow \mathbf{K}) \to (\mathbf{Graph} \downarrow \mathbf{R})$ defined as $r^>(\langle Y, h^* \rangle) = \langle Y, r \circ h^* \rangle$ on objects and $r^>(k^*) = k^*$ on arrows. As a consequence, each partial graph morphism $L \xleftarrow{l} K \xrightarrow{r} R$ induces a functor $\langle s \rangle : (\mathbf{Graph} \downarrow \mathbf{L}) \to (\mathbf{Graph} \downarrow \mathbf{R})$ defined as $\langle s \rangle = r^> \circ l^<$. Moreover $\langle \_ \rangle : \mathbf{Graph}^{\mathbf{P}} \to \mathbf{Cat}$ becomes a functor if we define it as $\langle G \rangle = (\mathbf{Graph} \downarrow \mathbf{G})$ on objects.

A *typed graph rule* in the double pushout sense is a typed partial graph morphism where the right-hand side is injective. The class of all typed graph rules is denoted by $\mathbf{Rules}$. Then a *typed graph grammar* $\mathbf{G} = (TG, P, \pi)$ consists of a type graph $TG \in \mathbf{Graph}$, a set of production names P, and a mapping $\pi : P \to \mathbf{Rules}$ associating with each production name its rule; if $p \in P$, $\pi(p)$ is also called the *sort* of p. A *graph grammar morphism* $f : \mathbf{G}_1 \to \mathbf{G}_2$ from typed grammar $\mathbf{G}_1 = (TG_1, P_1, \pi_1)$ to grammar $\mathbf{G}_2 = (TG_2, P_2, \pi_2)$ is a pair $f = (f_{TG}, f_p)$ where $f_{TG} : TG_1 \to TG_2$ is a partial graph morphism, and $f_P : P_1 \to P_2$ is a mapping of production names such that the sort of productions is preserved, i.e., $\pi_2(f_P(p)) = \langle f_{TG} \rangle(\pi_1(p))$ for all $p \in P_1$. (Here the functor $\langle f_{TG} \rangle$ is extended to arbitrary diagrams.) The category $\mathbf{GraGra}^+$ has typed graph grammars as objects and graph grammar morphisms as arrows. Composition and identities are defined componentwise.

As anticipated above, the category **GraGra**$^+$ just introduced is a simplified version of category **GraGra** of [1], because our typed grammars do not have a start graph, and the type component f_{TG} of grammar morphisms must be a partial graph morphism instead of an arbitrary span (note that **GraGra**$^+$ is not a subcategory of **GraGra**). It is worth stressing that the elimination of start graphs is a necessary condition to show the co-completeness of **GraGra**$^+$, that is proved below, because otherwise a counter-example to co-completeness can be obtained easily by adapting a similar negative result for marked P/T Petri nets [8]. On the other hand, the restriction imposed on grammar morphisms avoids the assumption of an "associative choice of pullbacks" made in [1] in order to ensure the well-definedness of span composition, making the overall technical treatment easier. It is still an open question whether category **GraGra**$^+$ remains co-complete (and under which conditions), if we allow for the more general morphisms of category **GraGra**.

Proposition 2.1 *The category* **GraGra**$^+$ *is finitely co-complete.*

Proof (Sketch) The empty graph grammar is initial in **GraGra**$^+$ because the empty (type) graph is initial in **Graph**$^{\mathbf{P}}$ and the empty set (of production names) is initial in **Set**. Moreover **GraGra**$^+$ has all pushouts that are constructed component-wise in **Graph**$^{\mathbf{P}}$ and **Set** using the functor property of $\langle \_ \rangle$. □

In [1] the *free transition system* of a grammar is obtained by generating all derived productions, i.e., all the double pushout diagrams having a production on top. The name of a derived production contains all the information about the double pushout, and its sort is the bottom span of the diagram. A morphism between graph transition systems is a grammar morphism that preserves derived productions, and the resulting category is denoted by **GraTS**. The obvious forgetful functor U : **GraTS** $\rightarrow$ **GraGra**, that regards every graph transition system as a graph grammar forgetting the additional structure of derived productions, has a left adjoint TS : **GraGra** $\rightarrow$ **GraTS** associating with each grammar its free transition system. Furthermore, the *free derivation system* of a grammar **G** is constructed by closing the set of productions **G** not only under derived productions, but also under sequential composition. The forgetful functor V, regarding every derivation system in **GraCat** as a graph grammar, has a left adjoint, too, that assigns to each grammar its free derivation system.

These results of [1] can be transferred *verbatim* to our simplified categories **GraGra**$^+$, **GraTS**$^+$, and **GraCat**$^+$, where the last two are obtained from **GraTS** and **GraCat**, respectively, by eliminating start graphs and by imposing the expected restriction on arrows.

Proposition 2.2 (i) *The forgetful functor* U : **GraTS**$^+$ $\rightarrow$ **GraGra**$^+$ *has a left adjoint* TS : **GraGra**$^+$ $\rightarrow$ **GraTS**$^+$.

(ii) *The forgetful functor* V : **GraCat**$^+$ $\rightarrow$ **GraGra**$^+$ *has a left adjoint* DS : **GraGra**$^+$ $\rightarrow$ **GraCat**$^+$.

Proof (Sketch) The type graph component f_{TG} of a grammar morphism is

left unchanged by the free constructions corresponding to functors TS and DS, as defined in [1], as well as by the forgetful functors. Thus all the mentioned functors preserve our restriction to partial graph morphisms. □

As a consequence **GraGra**$^+$ morphisms preserve direct derivations and derivation sequences as well as independence of direct derivations (see [1]).

3 Structuring of Typed Graph Grammars

One main advantage of having defined a category of graph grammars is that standard categorical constructions may be used to model suitable operations on grammars. In particular, colimits in **GraGra**$^+$, that are shown to exist in Proposition 2.1, can be used to compose graph grammars with respect to common subparts. The use of colimits to model the gluing of systems with shared subsystems is very common (and it is well motivated, for example, in [3]), and has the immediate advantage that the semantic functor is compositional with respect to such operations, by general categorical results. We state this property for pushouts (which is the kind of composition we shall use), but it holds for arbitrary colimits.

Proposition 3.1 (compositionality of semantics w.r.t. structuring) *Let* $\mathbf{G}_0, \mathbf{G}_1, \mathbf{G}_2$ *be typed graph grammars and* $\mathbf{G}_3$ *be the union defined as the pushout of* $\mathbf{G}_1 \xleftarrow{f_1} \mathbf{G}_0 \xrightarrow{f_2} \mathbf{G}_2$ *in* **GraGra**$^+$. *Then the semantics* $DS(\mathbf{G}_3)$ *of the union coincides with the union of the semantics defined as the pushout of the translated diagram* $DS(\mathbf{G}_1) \xleftarrow{DS(f_1)} DS(\mathbf{G}_0) \xrightarrow{DS(f_2)} DS(\mathbf{G}_2)$ *in* **GraCat**$^+$.

Proof. The semantic functor DS : **GraGra**$^+$ $\rightarrow$ **GraCat**$^+$ preserves pushouts (and all colimits) because it is a left adjoint functor. □

We shall show now, as an example, how a graph grammar specifying some basic operations on a list of lists can be obtained by taking the pushout of two disjoint copies of a grammar for lists with respect to a suitable subgrammar. Figure 2 shows the grammar $\mathbf{G}_{list} = (TG_{list}, \{new, ins, add\text{-}data, remove, gc\}, \pi_{list})$ which implements some operations on lists of elements of a datatype D which is not further specified. According to the definitions, TG_{list} is an unlabeled graph, thus the names written near arcs and nodes (that are depicted as circles) are their identities. Nodes of TG_{list} are the types of the basic components of a list, while arcs describe the way they can be related. LP, for *List Pointer*, is the (type of the) pointer to a list, and can have either a *nil* loop (the list does not exist), or can point to the first *List Element* (LE) of the list through a *fst* arc. Notice that loops are depicted as rounded arcs pointing to the node that is both source and target. A list element can either be the last one (if it has a *last* loop) or it has a *next* list element, and in this case it may carry one data element. If the data is present, it is pointed by a *data* arc, otherwise the list element has a *null* loop. A data element is simply a node labeled by D with a *dummy* loop, which is not further specified in this grammar. The sorts of the production names of $\mathbf{G}_{list}$ are drawn as partial morphisms in category TG_{list}-**Graph** (instead of as spans), where nodes belonging

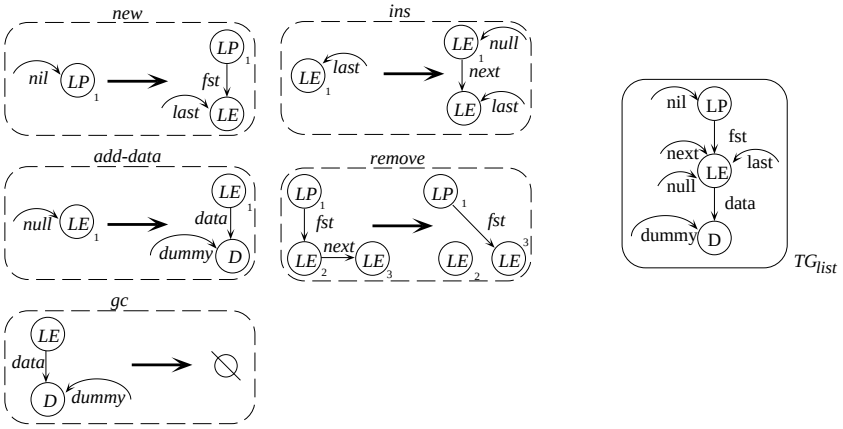

Fig. 2. Grammar $\mathbf{G}_{list}$, modeling manipulation of lists.

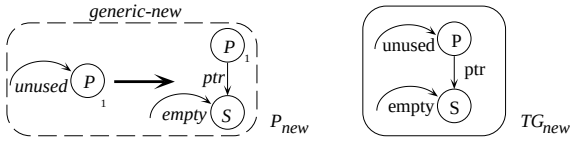

Fig. 3. Grammar $\mathbf{G}_{new}$ implementing a generic *new* operation.

to the domain and their images are marked with the same natural number. The typing morphisms are indicated by labeling each item of a graph with the name of its image in TG_{list}, written in italics. Production *new* creates an empty list (having only one list element which carries no data and is marked as *last*) from an unused list pointer; *ins* inserts a new list element at the end of a list; *add-data* adds a data element D to a list element carrying no data; *remove* eliminates the first element of the list by changing the *fst* pointer from LP (thus a FIFO strategy is ensured), and leaving the skipped list element as "garbage"; and *gc* performs garbage colletion deleting a list element and the attached data. Note that since we are using the double pushout approach, the application of a production to an occurrence morphism is subject to the *gluing conditions* [5]; thus production *gc* cannot be applied if some other arc is connected to the LE node, because the *dangling condition* would not be satisfied. This fact guarantees that only garbage is deleted.

Now, note that productions *new* and *add-data* are isomorphic but for the labeling: Indeed, both model the creation of a new, empty data structure (the list in *new*, the unspecified data in *add-data*). The idea is to obtain a grammar modeling the manipulation of lists of lists by gluing together two instances of grammar $\mathbf{G}_{list}$, and identifying the *add-data* production of the first with the *new* production of the second. Such an identification can be obtained by considering another grammar, $\mathbf{G}_{new}$, depicted in Figure 3, having only one production *generic-new* that given an *unused* pointer P, creates a new *empty* structure S and connect it to P through a *ptr* arc.

Let $\mathbf{G}'_{list}$ be a copy of the grammar for lists of Figure 2, but where all names have a prime. Then Figure 4 shows the pushout in category $\mathbf{GraGra}^+$ of the two grammar morphisms $f_1 : \mathbf{G}_{new} \to \mathbf{G}_{list}$ and $f_2 : \mathbf{G}_{new} \to \mathbf{G}'_{list}$. Morphisms f_1 and f_2 are specified by giving the mappings of the (various items of the)

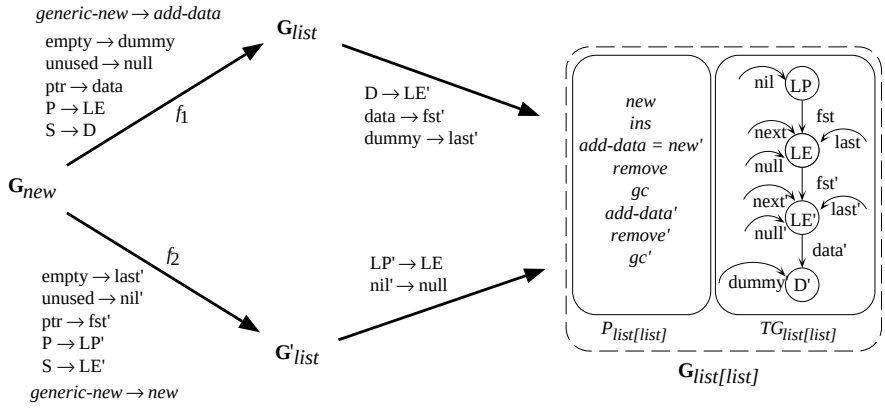

Fig. 4. A pushout diagram in category $\mathbf{GraGra}^+$ defining grammar $\mathbf{G}_{list[list]}$.

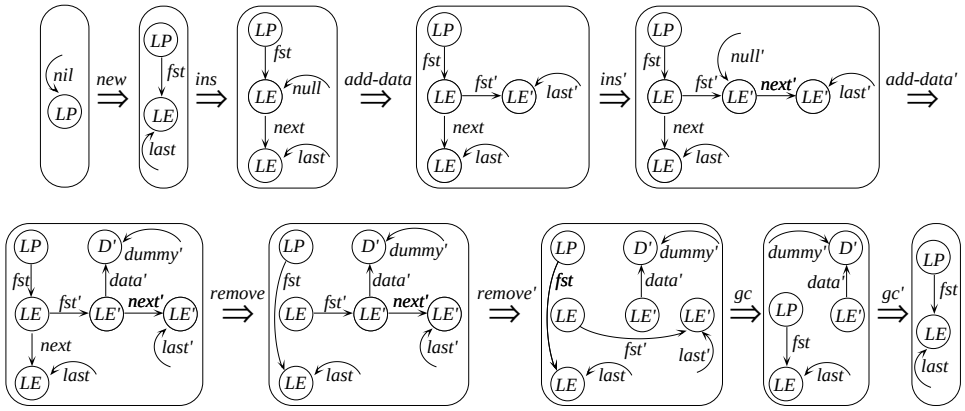

Fig. 5. A sample derivation for grammar $\mathbf{G}_{list[list]}$.

type graph, and of the only production; the required commutativity properties can be checked easily. The right part of the figure shows the grammar $\mathbf{G}_{list[list]}$ resulting from the pushout, and the two injection morphisms from the component grammars. The type graph $TG_{list[list]}$ is defined only up to isomorphism, and the injections are specified only for the items for which they are not the identity. The productions of $\mathbf{G}_{list[list]}$ are not depicted but only their names are listed: they can be obtained from the corresponding productions in the component grammars, by changing the labels of the graphs items according to the injection morphism. Note that $\mathbf{G}_{list[list]}$ has seven productions (and not eight) because productions *add-data* and *new'* are identified by the pushout construction, as expected.

Figure 5 shows a sample derivation in grammar $\mathbf{G}_{list[list]}$, showing the effect of the various productions: Such a derivation belongs to the derivation system $DS(\mathbf{G}_{list[list]})$, which is a small category that can be obtained, by Proposition 3.1, as the pushout in category $\mathbf{GraCat}^+$ of functors $DS(f_1) : DS(\mathbf{G}_{new}) \to DS(\mathbf{G}_{list})$ and $DS(f_2) : DS(\mathbf{G}_{new}) \to DS(\mathbf{G}'_{list})$.

4 Refinement of Typed Graph Grammars

The graph grammar morphisms introduced in Section 2 are quite general. In our view, it is not yet completely clear what is the relationship between

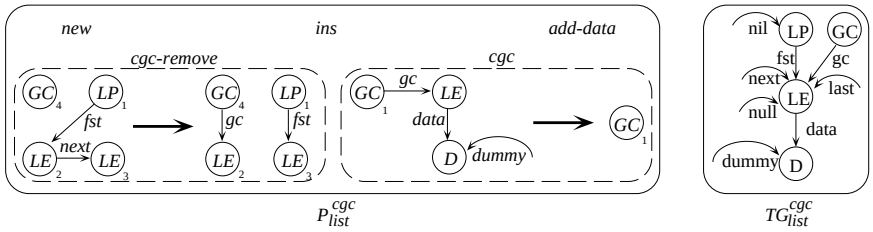

Fig. 6. Grammar $\mathbf{G}^{cgc}_{list}$, a refinement of $\mathbf{G}_{list}$ implementing centralized garbage collection.

two grammars when there exists a morphism $f : \mathbf{G}_1 \to \mathbf{G}_2$, except that all the derivations of $\mathbf{G}_1$ can be mapped to corresponding derivations of $\mathbf{G}_2$. In this section we will show with an example that, at least in certain cases, such a morphism indicates that $\mathbf{G}_1$ is a *refinement* of $\mathbf{G}_2$, in the sense that it implements the same functionalities, but the involved data structures are more complex. Clearly, not all grammar morphisms correspond to refinements in this sense. For example, grammar $\mathbf{G}_{new}$ cannot be considered at all as a "refinement" of $\mathbf{G}_{list}$. The following definition narrows the class of grammar morphisms eligible as refinements, accordingly with the informal requirements just given.

Definition 4.1 (refinement morphisms) *Given a graph grammar morphism $r = (r_P, r_{TG}) : \mathbf{G}_1 \to \mathbf{G}_2$, we say that $\mathbf{G}_1$ is a* refinement *of $\mathbf{G}_2$ if both the partial graph morphism $r_{TG} : TG_1 \to TG_2$ and the function $r_P : P_1 \to P_2$ are surjective. In this case r is called a* refinement morphism.

Surjectivity guarantees that $\mathbf{G}_1$ has all the functionalities of $\mathbf{G}_2$, but since morphisms r_{TG} can be partial, it may handle more refined data structures. We consider $\mathbf{G}_1$ as a refinement of $\mathbf{G}_2$ and not vice versa, in order to allow the refinement of one type of $\mathbf{G}_1$ by several ones of $\mathbf{G}_2$ (in case that f_{TG} is not injective).

As an example, we present grammar $\mathbf{G}^{cgc}_{list}$, which is a refined version of grammar $\mathbf{G}_{list}$ where *centralized garbage collection* is implemented, in the sense that a pointer is kept to each list element that becomes garbage. Figure 6 shows the new grammar. The type graph is obtained by adding to TG_{list} a node named GC, and an arc named gc pointing to node LE. The productions *new*, *ins*, and *add-data* are identical to the corresponding ones for $\mathbf{G}_{list}$, and are not depicted. There is an obvious morphisms $r : \mathbf{G}^{cgc}_{list} \to \mathbf{G}_{list}$ which forgets node GC and arc gc of TG^{cgc}_{list}, maps production names *cgc-remove* and *cgc* to *remove* and *gc*, respectively, and is the identity on the other names. In fact, if from productions *cgc-remove* and *cgc* we remove all items labeled by GC and gc, we obtain the productions *remove* and *gc* of $\mathbf{G}_{list}$. Morphisms r is clearly a refinement, because it is surjective. Let us show now that this notion of refinement is compatible with the structuring mechanisms of the previous section. We have the following easy result (that holds not only for pushouts, but also for arbitrary colimits).

Proposition 4.2 (compatibility of structuring and refinement) *Consider the diagram of Figure 7 (a) in category* **GraGra**$^+$*, where $\mathbf{G}_3$ is*

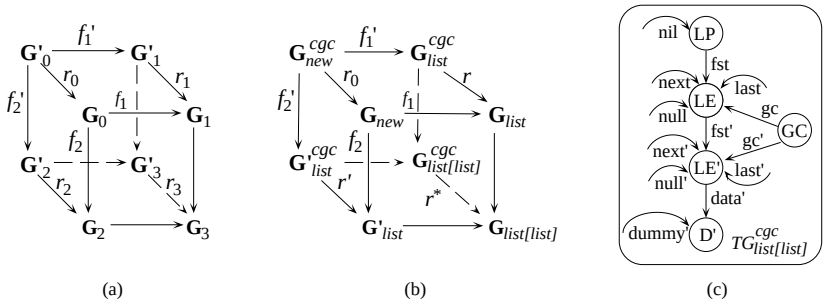

Fig. 7. (a) Compatibility of structuring and refinement. (b) An example. (c) The type graph of the resulting grammar.

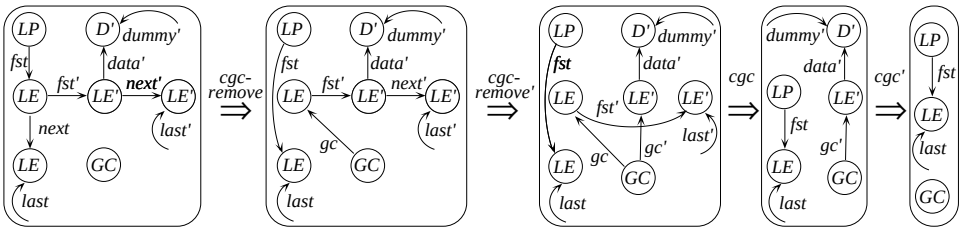

Fig. 8. A sample derivation using centralized garbage collection.

the pushout object of f_1 and f_2, and $r_i : \mathbf{G}'_i \to \mathbf{G}_i$ are compatible *refinement morphisms, for $i = 0, 1, 2$ (i.e., the top and left squares commute). Then there is a unique refinement morphism $r_3 : \mathbf{G}'_3 \to \mathbf{G}_3$ making all the diagram commute, where $\mathbf{G}'_3$ is the pushout object of f'_1 and f'_2.*

Proof (Sketch) The existence and uniqueness of morphism r_3 follows by the universal property of pushouts. The fact that it is surjective both on productions and on the type graph follows by the surjectivity of the other refinement morphisms and by the fact that the injections in the pushout object $\mathbf{G}_3$ are jointly surjective. □

As an example, let us show how grammar $\mathbf{G}_{list[list]}$ can be refined to a grammar $\mathbf{G}^{cgc}_{list[list]}$ implementing centralized garbage collection, by just specifying the refinement of the component grammars. In the diagram of Figure 7 (b), the front square is the pushout of grammars of Figure 4; r is the refinement morphisms presented above, and r' is a similar morphism relating disjoint copies of the same grammars. Furthermore, grammar $\mathbf{G}^{cgc}_{new}$ is obtained by adding to TG_{new} a single node called GC in the type graph; morphisms f'_1 and f'_2 are like f_1 and f_2, but they additionally map the GC node of TG^{cgc}_{new} to the GC and GC' nodes of TG^{cgc}_{list} and $TG^{cgc'}_{list}$, respectively. By Proposition 4.2 there exists only one grammar morphism r^* from the pushout object of f'_1 and f'_2, $\mathbf{G}^{cgc}_{list[list]}$, to $\mathbf{G}_{list[list]}$ such that the diagram commutes, and moreover r^* is a refinement. Figure 7 (c) shows the type graph of grammar $\mathbf{G}^{cgc}_{list[list]}$, and Figure 8 shows a derivation for that grammar that refines the second part of the derivation of Figure 5. Note that there are still two distinct rules for garbage collection, but both use the same GC-labeled node, that can be considered as a global repository for pointers to garbage data.

5 Conclusions and Future Work

After summarizing the categorical semantics of graph grammar proposed in [1], we discussed the use of colimits in the category of graph grammars as a structuring mechanism for the specification of large grammars, showing that the categorical semantics is automatically compositional with respect to such mechanism. Furthermore, we showed that certain morphisms of grammars may be interpreted as a refinement relation (where the source grammar refines the target one), and proved that such notion of refinement is compatible with the structuring mechanisms.

Our notion of refinement applies to the data of the specification, i.e. the type graph, while the refinement of operations (rules) is more or less fixed by the data refinement. To model operation refinement one has to map a single rule to a derived rule, representing a compound operation. This however requires more general graph grammar morphisms. Moreover, one may ask that the refined grammar implements more functionalities, as in the case of the inheritance relation among classes in object oriented systems. In this case it would be no more true that a derivation in the source grammar can always be mapped to a derivation in the target grammar. We believe that this could be modeled by allowing in a graph grammar morphism a *partial* function among productions.

References

[1] A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and J. Padberg, *Typed graph grammars and their adjunction with categories of derivations*, submitted for publication.

[2] A. Corradini, U. Montanari, and F. Rossi, *Graph processes*, to appear in *Fundamenta Informaticae*, 1995.

[3] J.A. Goguen, *A categorical manifesto*, *Math. Struc. Comput. Sci.* **1** (1991).

[4] H. Ehrig and R. Bardohl, *Specification techniques using dynamic abstract data types and application to shipping software*, Proc. of the International Workshop on Advanced Software Technology, 1994, 70–85.

[5] H. Ehrig, *Introduction to the algebraic theory of graph grammars*, in V. Claus, H. Ehrig, and G. Rozenberg (Eds), *Proceedigs of the 1st International Workshop on Graph Grammars and Their Application to Computer Science and Biology*, Lecture Notes in Computer Science 73, 1979, 1–69.

[6] M. Korff, *Graph-interpreted graph transformations for concurrent object-oriented systems*, submitted for publication.

[7] M. Löwe, *Algebraic approach to single-pushout graph transformation*, *Theoret. Comput. Sci* **109** (1993) 181–224.

[8] J. Meseguer and U. Montanari, *Petri nets are monoids*, *Information and Computation*, **88** (1990) 105–155.