A Graph Formalism for Ordered Edges

Maarten de Mol* and Arend Rensink

http://www.cs.utwente.nl/~molm, molm@cs.utwente.nl
http://www.cs.utwente.nl/~rensink, rensink@cs.utwente.nl
Department of Computer Science, University of Twente
Enschede, The Netherlands

Abstract: Though graphs are flexible enough to model any kind of data structure in principle, for some structures this results in a rather large overhead. This is for instance true for lists, i.e., edges that are meant to point to an ordered collection of nodes. Such structures are frequently encountered, for instance as ordered associations in UML diagrams. Several options exist to model lists using standard graphs, but all of them need auxiliary structure, and even so their manipulation in graph transformation rules is not trivial.

In this paper we propose to enrich graphs with special ordered edges, which more naturally represent the intended structure. We show that the resulting graphs still form an adhesive HLR category, and so the standard results from algebraic graph transformation apply. We show how lists can be manipulated. We believe that in a context where lists are common, the cost of a more complicated graph formalism is outweighed by the benefit of a smaller, more appropriate model and more straightforward manipulation.

This technical report is the extended version of [MR10].

Keywords: Graph Rewriting, Ordered Edges

1 Introduction

The context of the work in this paper is graph transformation. This means that we use graphs, essentially only consisting of nodes and edges, to model different kinds of structures such as real-world systems or software concepts. A rich source of such structures comes from software engineering, in the form of UML models. Graph transformation offers a mathematically well-founded method for systematically encoding changes to graphs; this in turn can be used to describe the dynamics of the system being modelled.

In principle, appropriate compositions of the basic building blocks of nodes and binary edges can encode arbitrary structures. In many cases the resulting graphs reflect the original structures quite naturally. There are, however, situations in which the encoding is awkward, for instance because it requires auxiliary elements in the graph that do not directly reflect anything from the original structure. This impacts the understandability and complexity of the encoding, and thus decreases the usability of graph transformation. In such cases, one may choose to use a richer graph formalism instead, which more closely reflects the structures at hand.

* Supported by the EU Artemis project CHARTER
Examples of enriched graph formalisms, introduced exactly for the reason of modelling particular kinds of structures more naturally, are: attributed graphs [EEPT06a], hierarchical graphs [DHP02], and hypergraphs [Hab92]. There is a price to pay for such enrichments, in the form of added complexity in their usage and understanding (often called the learning curve), as well as in their manipulation, both on the level of theory and of implementation. It follows that enrichments in the graph formalism are only justified if the resulting increase in complexity is outweighed by the corresponding advantages in modelling.

Fortunately, on the theoretical level there is a touchstone against which we can measure the feasibility of any enriched graph formalism: namely, the concept of adhesive HLR categories, developed in [EPPH06] — see [EEPT06b] for an extensive description. That is, if one can prove that an enriched notion of graphs satisfies the constraints of HLR-adhesiveness, then there is a standard way to define their transformation, and many desirable properties are known to hold.

In this paper we propose an enrichment of the basic graph formalism to cope with the structural concept of ordered lists. Such lists occur frequently in practice, for instance in the form of ordered associations in UML diagrams or array- and list-like structures in software. We will argue that encoding lists using simple graphs introduces spurious elements and thus increases their complexity; also the manipulation of the encodings is non-trivial. Thus, in line with the reasoning above, it makes sense to solve the problem of modelling list structures by enriching the graphs. In order to justify the cost of a more complex formalism on the level of theory, we show that the resulting category of graphs is still adhesive HLR.

In the next section, we motivate and explain our extension on an intuitive level, using an example inspired by the Olympic winter games. After that, Section 3 presents the formal definitions and states the main theoretical result (HLR adhesiveness of the resulting category). We show the use of list graphs in Section 4. Finally, Section 5 discusses related work and presents conclusions.

2 Motivation

As a motivating example, we use sporting events taking place in the 2010 Olympic winter games. In particular, we concentrate on ice-skating. Before the games, every skating event has a list of participants; the order in the list corresponds to their starting order at the event. For instance, Figure 1 shows two events (1500 m and 5 km for men). The notation means that the 1500 m event has four participants, in the order from top to bottom, whereas the 5 km event has three, namely Kramer, Tuitert and Davis (in that order). A third event, the 10 km, has an empty list of participants.

Intuitively straightforward operations one may want to perform on such a list are:

- Appending an element when a new participant is enrolled;
- Removing participants convicted of doping abuse;
- After the event, moving the winner to the top of the list.

A more complex operation is list reversal. For instance, if we started with a ranking list (in which the seasonal best skater is at the top), then it needs to be reversed to get the starting order.
Figure 1: Two skating events with overlapping lists of participants

2.1 Plain graph encoding

There are several ways to encode such lists using plain graphs, consisting only of nodes and binary edges. We discuss the main issues.

- The core problem is to specify the order of the elements. For this purpose, one can either rely on an implicit ordering, for instance using indices, or introduce an explicit ordering using special edges. Indices require updating whenever elements are added or removed (except at the end of the list).

- Elements can be shared among lists (as Figure 1 shows), or may even occur multiple times in the same list. For this reason, the indices or special edges specifying the ordering cannot be incident to the list elements themselves (this would introduce confusion between the lists); rather, one needs an intermediate layer of “slot” nodes.

- It is often convenient, or even necessary, to express that a given element is in a particular list. To encode this information, we need further special edges pointing from the list owner to the elements, or vice versa.

- Many list operations explicitly refer to the first or last element. To express this, either we need negative application conditions stating that the element has no predecessor, respectively successor; or this information can be captured using special edges — which, however, then have to be maintained while manipulating the list.

- The empty list needs to be represented in some special way, as in that case there are no element or slot nodes to attach information to.

Clearly, such a graph representation is expensive, in the sense of requiring many auxiliary elements; moreover, unless one is careful, the last two issues will require case distinctions in transformation rules.

From programming, we know an encoding for lists that copes with most of the issues relatively well (in particular avoiding case distinctions), but is expensive in terms of overhead: namely, a circular linked list consisting of “slot” nodes pointing to the elements and back to the list owner, and a special “head” node without an element, marking the start and the end of the list. Figure 2 shows a plain graph encoding of the structure of Figure 1.
Figure 2: Plain graph representation of the structure in Figure 1

Figure 3: Plain graph rule moving the winner of an event to the top of the list.

Figure 3 shows an example rule that will result in the winner of an event being moved to the start of the list. The figure shows the left hand side and right hand side of the rule; the connecting morphisms are implicit in the positioning of the nodes. The unlabelled nodes are meant to match any node in the graph; in particular, they may match Head or Slot nodes. A solution that works for properly typed graphs requires inheritance. Note that this only works under the assumption that non-injective matches are allowed.

An important observation is that the issues discussed above are exactly those one encounters while programming with lists. This goes against the idea that graph transformation provides an abstract, declarative way of manipulating structures. To name one consequence, if the graph model is used for the design of a software system, from which an implementation is to be derived, then the graph representation choices will influence the implementation, possibly in unintended ways. For instance, the encoding in Figure 2 makes it unnatural to choose an array-based implementation.
2.2 List edges

The proposal in this paper is to enrich graphs with explicit support for lists, avoiding both the overhead and the “programming” nature of the plain graph encoding. We do this by extending the notion of edges: rather than binary edges with a single source node and a single target node, we propose to use list edges of which the target is a sequence of nodes. Thus, list edges are somewhat like hyperedges in that they may have different numbers of tentacles: however, hyperedges typically have a fixed number of tentacles (called the arity) determined by their labels, which is not the case for list edge arity.

For instance, Figure 1 is a straightforward visualisation of a graph with list edges from the Event nodes to different sequences of Part nodes. The string of “knots” in the edge gives the order of the elements in the list; the arrows from the knots point to the actual elements.

The real innovation, however, does not lie in the graphs but in the rules. For these, we introduce a new type of node, called list nodes, which will only appear in rules and can be seen as standing for arbitrary sequences of nodes from the host graph. List nodes can only occur within edge targets, never as sources. Graph morphisms are extended to list nodes as follows: every list node is matched either by a sequence of plain nodes, or by a single list node. This is extended to list edges in the natural way.

For instance, Figure 4 shows the rule performing the same operation as the one in Figure 3, but this time for list graphs. The ‘doubled’ nodes are list nodes. The parts edge in the left hand side matches any list edge in the host graph from an Event node, pointing to an arbitrary sequence of nodes (matched by the upper list node of the LHS), followed by the Part-node that the winner-edge points to, followed by another arbitrary sequence of nodes (matched by the lower list node of the LHS). The effect of the rule is to delete this list edge and create a new one, in which the Part-node and the first sub-sequence are swapped. This has the effect of moving the Part-node to the top of the list.

An example of the application of this rule is shown in Figure 5. The initial state is the same as in Figure 1, but now with Kramer and Tuitert indicated as winners for the 1500m and 5000m respectively. The rule can be applied twice, resulting in the right hand side graph.

3 Formalisation

In this section, we will show that lists can be incorporated in graph theory in a sound manner. For this purpose, we will extend a standard representation of multi-sorted graphs with list nodes and list edges, and we will show that the result is an adhesive HLR category [EPPH06]. We will
use double push-outs (DPO) for the formalisation of graph rules.

First, we extend a standard \((V,E,src,tgt,lab)\) representation of multi-sorted graphs, by: (1) splitting \(V\) into \(\hat{V}\) (normal nodes) and \(\bar{V}\) (list nodes); and (2) changing the result of \(tgt\) from \(V\) (a single node) to \(V^*\) (a sequence of nodes, may be empty). In other words, we add list nodes and replace one-to-one (plain) edges with one-to-many (list) edges:

**Definition 1 (multi-sorted list graphs)**

Let \(G = (\hat{V}, \bar{V}, E, src, tgt, lab)\) be a multi-sorted list graph, where:

- \(\hat{V}\) and \(\bar{V}\) are the sets of plain nodes and list nodes respectively (let \(V\) denote \(\hat{V} \cup \bar{V}\))
- \(E\) is the set of (list) edges
- \(\hat{V}\), \(\bar{V}\) and \(E\) are disjoint
- \(src: E \rightarrow \hat{V}\) is the function that yields the source node of an edge
- \(tgt: E \rightarrow V^*\) is the function that yields the sequence of target nodes of an edge
- \(lab: E \rightarrow L\) is the labelling function (assuming a fixed set of labels \(L\))

As usual, we will use graph homomorphisms as arrows in our to be defined category. A homomorphism \(f : G \rightarrow H\) is a structure preserving mapping of nodes and edges. In our case, three mappings have to be defined: (1) one for plain nodes, which are mapped to plain nodes; (2) one for list nodes, which are mapped either to list nodes or to sequences of plain nodes; and (3) one for list edges, which are mapped to list edges. The one-to-one mapping of list nodes will be used to restrict our graph rules, and the one-to-many mapping of list nodes will be used for the matching of a rule to a graph.

For the sake of convenience, we will combine the mappings of nodes into a single function that always produces a sequence. In the case of plain nodes, the mapping will always produce a singleton sequence. For converting a singleton sequence to its element, we will use the auxiliary function \(f^-\). Furthermore, we will use the auxiliary function \(f^+\) to lift a function to sequences.

**Definition 2 (lifting of functions)**

(a) For all sets \(A\) and \(B\) and functions \(f: A \rightarrow B^*\), let \(f^-: A \rightarrow B\) be the partial function that is defined by \(f^-(a) = b\) if \(f(a) = \langle b \rangle\).

(b) For all sets \(A\) and \(B\) and functions \(f: A \rightarrow B^*\), let \(f^+: A^* \rightarrow B^*\) denote the natural extension of \(f\) to sequences that is defined by \(f^+(\langle a_1 \ldots a_n \rangle) = f(a_1) \oplus \ldots \oplus f(a_n)\), where \(\oplus\) denotes concatenation of sequences.
Definition 3 (homomorphisms)
Let $G = (\hat{V}_G, V_G, E_G, src_G, tgt_G, lab_G)$ and
$H = (\hat{V}_H, V_H, E_H, src_H, tgt_H, lab_H)$ be multi-sorted list graphs.
Let $f = (f_V, f_E)$ with $f_V : V_G \rightarrow V_H$ and $f_E : E_G \rightarrow E_H$ map the nodes and edges of $G$ to $H$.
Then, $f$ is a homomorphism when the following conditions hold:
- for all $v_g \in \hat{V}_G$ there exists a $v_h \in \hat{V}_H$ such that $f_V(v_g) = \langle v_h \rangle$
- for all $v_g \in V_G$, there either exists a $v_h \in V_H$ such that $f_V(v_g) = \langle v_h \rangle$, or $f_V(v_g) \in \hat{V}_H^*$
- $lab_H \circ f_E = lab_G$
- $src_H \circ f_E = f_V \circ src_G$
- $tgt_H \circ f_E = f_V \circ tgt_G$

The composition of two homomorphisms can now easily be defined by means of a combination of function composition and natural extension to sequences. By construction, it follows that the result is a homomorphism as well, which allows us to define list graphs as a category.

Definition 4 (composition of homomorphisms)
If $f = (f_V, f_E) : G \rightarrow H$ and $g = (g_V, g_E) : H \rightarrow I$ are homomorphisms on list graphs, then $g \circ f$ is defined by $(g_V \circ f_V, g_E \circ f_E)$.

Definition 5 (list graphs as a category)
The category $\mathbb{G}L$ consists of list graphs (Definition 1) as objects, homomorphisms (Definition 3) as arrows and composition as in Definition 4. The identity arrows are the homomorphisms that are pairs of identity functions.

The next step is to show that the constructed $\mathbb{G}L$ is also an adhesive HLR category, which allows DPO graph rewriting to be defined in it. First, we briefly recall the definitions:

Definition 6 (van Kampen squares)
A pushout $p$ is a VK-square if for any commutative cube where the bottom face is $p$, it holds that the top face is a pushout iff the front faces are pullbacks.

Definition 7 (adhesive HLR categories)
A category $\mathcal{C}$ with a given subclass of $M$-morphisms is an adhesive HLR-category, iff:
- $M$ is a class of monomorphisms that is closed under isomorphisms, composition and decomposition
- all objects have pullbacks and pushouts along $M$-morphisms, and $M$-morphisms are closed under pushouts and pullbacks
- all pushouts form VK-squares

In order to show that $\mathbb{G}L$ is an adhesive HLR category, we have to identify a subclass of $M$-morphisms and prove the properties that are listed in Definition 7. This is accomplished by the following definitions and theorems. First, Definition 8 defines the class of $M$-morphisms. Definition 11 defines the pullbacks, and Theorem 1 proves its correctness. Definition 12 defines the pushouts, and Theorem 2 proves its correctness. The overall adhesive proof is given in Theorem 3. Definitions 9 and 10 introduce notation that is used in the proofs.
**Definition 8** (M-morphisms in $\mathbb{GL}$)

A monomorphism $f = (f_V, f_E) : G \rightarrow H$ in $\mathbb{GL}$ belongs to the subclass $M$ if for all $v_G \in V_G$ there exists a $v_H \in V_H$ such that $f_V(v_G) = (v_H)$. In other words: a $M$-morphism does not perform matching of list nodes to sequences, but maps them one-to-one to list nodes only.

**Definition 9** (morphisms as functions)

(a) The image of a morphism $m = (f_V, f_E)$ is defined by $I(m) = \{v\mid \exists l \in \text{Ran}(f_V)[v \in l]\} \cup \text{Ran}(f_E)$.

(b) The function interpretation of a morphism $m = (f_V, f_E)$ is defined by:

$$m_o = \{(v, w) \mid (v, (w)) \in f_V \} \cup f_E$$

The function interpretation restricts $m$ to a one-to-one mapping, which maps plain nodes to plain nodes, list nodes to list nodes and edges to edges. If $m$ is a $M$-morphism, then $m_o$ models the full behavior of $m$. In that case, $m$ is also injective, and can be inverted for all $x \in I(m)$ (i.e. $m^{-1}o(x)$ is defined for all $x \in I(m)$).

(c) If $G = (\hat{V}, \hat{E}, \text{src}, \text{tgt}, \text{lab})$ is a subgraph of $H$, then the morphism $\text{embed}_{G:H} : G \rightarrow H$ is defined by $\{(v, (v)) \mid v \in \hat{V} \cup \hat{E}, (e, e) \mid e \in E\}$.

**Definition 10** (graphs as sets)

(a) The element set of a graph $G = (\hat{V}, \hat{E}, \text{src}, \text{tgt}, \text{lab})$ is defined by $ES(G) = \hat{V} \cup \hat{E} \cup E$.

(b) If $G$ is a graph, then the notation $x \in G$ abbreviates $x \in ES(G)$.

(c) The restriction of a graph $G = (\hat{V}, \hat{E}, \text{src}, \text{tgt}, \text{lab})$ to a set $X$ is defined straightforwardly by $G|_X = (\hat{V} \cap X, \hat{E} \cap X, \text{src}|_X, \text{tgt}|_X, \text{lab}|_X)$. Note that $G|_X$ is not automatically a graph itself. If $m$ is a morphism with $\text{Dom}(m) = G$, then $G|_{I(m)}$ is a valid graph, and also a subgraph of $G$. If $m$ is a $M$-morphism, then $G|_{I(m)}$ is isomorphic to $\text{Cod}(m)$.

**Definition 11** (construction of pullbacks)

Let $B \xrightarrow{b} A \xleftarrow{c} C$ be a cospan, where $b$ is a $M$-morphism and $B$ and $C$ are disjoint.

Then, the pullback is the span $B \xrightarrow{\text{eqC}} D \xleftarrow{\text{embed}_{D\leftarrow C}} C$, which is defined by the following steps:

- Define $D$ to be the largest subgraph of $C$ such that all elements of $D$ are mapped to elements of $A|_{I(b)}$ by $c$. Note that $\text{embed}_{D\leftarrow C}$ (see Definition 9(c)) maps $D$ to $C$.
- Define $c'$ to be the morphism that maps $D$ to $A|_{I(b)}$ by means of $c$.
- Define $z$ to be the morphism that maps $A|_{I(b)}$ to $B$ by means of $b^{-1}$. This morphism exists, because $b$ is a $M$-morphism and $A|_{I(b)}$ is isomorphic to $B$ (see Definition 10(c)).

**Theorem 1** (correctness of pullback construction)

The $B \xrightarrow{\text{eqC}} D \xleftarrow{\text{embed}_{D\leftarrow C}} C$ of Definition 11 is the pullback of $B \rightarrow A \leftarrow C$.

Proof, part 1 (diagram converges):

Let $a$ abbreviate $\text{embed}_{\hat{X}|_{I(b)}X}$. Observe that by construction of $D$, $c'$ and $z$ in Definition 11, the following equalities hold: [eqA] $a \circ c' = c \circ \text{embed}_{D\leftarrow C}$ and [eqB] $b \circ z = a$.

This implies [eqC] $c \circ \text{embed}_{D\leftarrow C} = [eqA] a \circ c' = [eqB] b \circ (z \circ c')$.

Proof, part 2 (unique mediator):

Assume $B \xrightarrow{X} C$ with [eqD] $b \circ y = c \circ x$. Define $x'$ to be the restriction of $x$ to the subgraph $D$ of $C$; then $x'$ is a morphism between $X$ and $D$ and $[eqE] \text{embed}_{D\leftarrow C} \circ x' = x$ holds.

Now, $x'$ is a mediating arrow of the pullback, because:

- Combining [eqC], [eqE] and [eqD] yields $b \circ z \circ c' \circ x' = [eqC] c \circ \text{embed}_{D\leftarrow C} \circ x' = [eqE] c \circ z \circ c'$.
\( x = [eqD] b \circ y \). Because \( b \) is monomorphic, this implies [eqF] \( (z \circ c') \circ x' = y \).

- embed\(_{D,C} \circ x' = x \) holds by [eqE]

Also, \( x' \) is unique, since any other \( q : X \to D \) with embed\(_{D,C} \circ q = x \) is isomorphic to \( x' \), because embed\(_{D,C} \) is monomorphic by construction (see Definition 9(c)).

\( \square \)

**Definition 12 (construction of pushouts)**

Let \( B \xleftarrow{b} A \xrightarrow{c} C \) be a span, where \( b \) is a \( M \)-morphism and \( B \) and \( C \) are disjoint.

Then, the pushout is the cospan \( B \xrightarrow{d} D \xleftarrow{e} C \), which is defined by the following steps:

- Define \( d = (d_V, d_E) \) (from \( B \) to \( D \)) to unify \( b^{-1} \) and \( c \) as follows:

\[
\begin{align*}
\cdot \quad d_V(v) &= \begin{cases} 
(v) & \text{if } v \notin I(b) \\
 c_V(b^{-1}_V(v)) & \text{if } v \in I(b)
\end{cases} & d_E(e) &= \begin{cases} 
 e & \text{if } e \notin I(b) \\
 c_E(b^{-1}_E(e)) & \text{if } e \in I(b)
\end{cases}
\end{align*}
\]

- Define \( D \) to be the graph with elements \( ES(C) \cup (ES(B) \setminus I(b)) \) and with functions:

\[
\begin{align*}
\cdot \quad src_D(e) &= \begin{cases} 
 src_C(e) & \text{if } e \in ES(C) \\
 d_V( src_B(e) ) & \text{if } e \in ES(B) \setminus I(b)
\end{cases} \\
\cdot \quad tgt_D(e) &= \begin{cases} 
 tgt_C(e) & \text{if } e \in ES(C) \\
 d'_V( tgt_B(e) ) & \text{if } e \in ES(B) \setminus I(b)
\end{cases} \\
\cdot \quad lab_D(e) &= \begin{cases} 
 lab_C(e) & \text{if } e \in ES(C) \\
 lab_B(e) & \text{if } e \in ES(B) \setminus I(b)
\end{cases}
\end{align*}
\]

Note that these definitions are wellformed: \( b \) is a \( M \)-morphism and can therefore be reversed on \( I(b) \), and \( B \) and \( C \) are disjoint making \( ES(C) \) and \( ES(B) \setminus I(b) \) disjoint as well. Therefore, \( D \) defines a graph, and embed\(_{C,D} \) defines a morphism between \( C \) and \( D \). That \( d \) is morphism as well, will be verified in Theorem 2.

**Theorem 2 (correctness of pushout construction)**

The \( B \xrightarrow{d} D \xleftarrow{e} C \) of Definition 12 is the pushout of \( B \xleftarrow{b} A \xrightarrow{c} C \).

Proof, part 1 (\( d \) is a homomorphism):

The first two conditions on homomorphisms (see Definition 3) concern wellformedness of \( d_V \).

For \( v \notin I(b) \), \( d_V(v) = (v) \), and wellformedness is obvious. For \( v \in I(b) \), \( d_V(v) = c_V(b^{-1}_V(v)) \), and wellformedness is inherited from \( c_V \), because \( b^{-1}_V(v) \) is always of the same type as \( v \).

The remaining equality conditions are shown by the following case distinction:

- Assume \( e \in E_B \) and \( e \notin I(b) \). Then:

  1. \( lab_D(d_E(e)) = lab_D(e) = lab_B(e) \)
  2. \( src_D(d_E(e)) = src_D(e) = d'_V( src_B(e) ) \)
  3. \( tgt_D(d_E(e)) = tgt_D(e) = d'_V( tgt_B(e) ) \)

- Assume \( e \in E_B \) and \( e \in I(b) \). Then:

  4. \( lab_D(d_E(e)) = lab_D(c_E(b^{-1}_E(e))) = lab_A(b^{-1}_A(e)) = lab_B(e) \)
  5. \( src_D(d_E(e)) = src_D(c_E(b^{-1}_E(e))) = src_C(c_E(b^{-1}_E(e))) = c'_V( src_A(b^{-1}_A(e)) ) \)
  6. \( tgt_D(d_E(e)) = tgt_D(c_E(b^{-1}_E(e))) = tgt_C(c_E(b^{-1}_E(e))) = c'_V( tgt_A(b^{-1}_A(e)) ) \)

Combining (1) and (4) yields \( lab_D \circ d_E = lab_B \).
Combining (2) and (5) yields \( \text{src}_D \circ d_E = d_{\nabla} \circ \text{src}_B \).

Combining (3) and (6) yields \( \text{tgt}_D \circ d_E = d_{\nabla} \circ \text{tgt}_B \).

Proof, part 2 (diagram converges):

The convergence of the pushout diagram is shown by the following case distinction:

- Assume \( v \in V_B \). Then:
  
  \[
  (d \circ b)_{\nabla}(v) = d_{\nabla}(b_{\nabla}(v)) = c_{\nabla}(b_{\nabla}^{-1}(b_{\nabla}(v))) = c_{\nabla}(b_{\nabla}^{-1}(v)) = c_{\nabla}(v) = (\text{embed}_{C,D} \circ c)_{\nabla}(v) \]

- Assume \( e \in E_B \). Then:
  
  \[
  (d \circ b)_{\nabla}(e) = d_{\nabla}(b_{\nabla}(e)) = c_{\nabla}(b_{\nabla}^{-1}(b_{\nabla}(e))) = c_{\nabla}(e) = (\text{embed}_{C,D} \circ c)_{\nabla}(e) \]

Combining (1) and (2) yields \( d \circ b = \text{embed}_{C,D} \circ c \).

Proof, part 3 (unique mediator):

Assume a graph \( X \) and a co-span \( B \xrightarrow{g} X \xleftarrow{h} C \) such that \( h \circ c = g \circ b \). Define the mediator by:

\[
f_{\nabla}(v) = \begin{cases} 
  h_{\nabla}(v) & \text{if } v \in \text{ES}(C) \\
  g_{\nabla}(v) & \text{if } v \in \text{ES}(B) \setminus \{b \} 
\end{cases}
\]

\[
f_{\nabla}(e) = \begin{cases} 
  h_{\nabla}(e) & \text{if } e \in \text{ES}(C) \\
  g_{\nabla}(e) & \text{if } e \in \text{ES}(B) \setminus \{b \} 
\end{cases}
\]

Basically, \( f \) is the union of \( g \) and \( h \), which is a homomorphism from \( D \) to \( X \) because \( B \) and \( C \) are assumed to be disjoint. Furthermore, \( f \) is the mediating arrow of the pushout, because:

- Assume \( v \in V_C \). Then:
  
  \[
  (f \circ \text{embed}_{C,D})_{\nabla}(v) = f_{\nabla}(c_{\nabla}(\text{embed}_{C,D}v)(v)) = f_{\nabla}(v) = g_{\nabla}(v) \]

- Assume \( e \in E_C \). Then:
  
  \[
  (f \circ \text{embed}_{C,D})_{\nabla}(e) = f_{\nabla}(c_{\nabla}(\text{embed}_{C,D}e)(v)) = e_{\nabla}(v) = g_{\nabla}(e) \]

- Assume \( \forall v \in V_B \) and \( \forall v \notin I(b) \). Then:
  
  \[
  (f \circ d)_{\nabla}(v) = f_{\nabla}(c_{\nabla}(dv)(v)) = f_{\nabla}(v) = g_{\nabla}(v) \]

- Assume \( \forall v \in V_B \) and \( \forall v \in I(b) \). Then:
  
  \[
  (f \circ d)_{\nabla}(v) = f_{\nabla}(c_{\nabla}(b_{\nabla}^{-1}(dv)) = h_{\nabla}(c_{\nabla}(b_{\nabla}^{-1}(v))) = g_{\nabla}(b_{\nabla}(b_{\nabla}^{-1}(v))) = g_{\nabla}(v) \]

- Assume \( e \in E_B \) and \( e \notin I(b) \). Then:
  
  \[
  (f \circ d)_{\nabla}(e) = f_{\nabla}(d_{\nabla}(e)) = e_{\nabla}(e) = g_{\nabla}(e) \]

- Assume \( e \in V_B \) and \( e \in I(b) \). Then:
  
  \[
  (f \circ d)_{\nabla}(e) = f_{\nabla}(c_{\nabla}(b_{\nabla}^{-1}(e))) = h_{\nabla}(c_{\nabla}(b_{\nabla}^{-1}(e))) = g_{\nabla}(b_{\nabla}(b_{\nabla}^{-1}(e))) = g_{\nabla}(e) \]

Combining (1) and (2) yields \( f \circ \text{embed}_{C,D} = h \).

Combining (3),(4),(5) and (6) yields \( f \circ d = g \). Finally, \( f \) is unique, since any other \( q : D \rightarrow X \) with \( q \circ \text{embed}_{C,D} = h \) is isomorphic to \( f \), because \( \text{embed}_{C,D} \) is epimorphic by construction (see Definition 9(c)).

\[ \Box \]

**Theorem 3** \((\mathbb{GL} \text{ is an adhesive HLR category})\)

\(\mathbb{GL} \text{ is an adhesive HLR category.}\)

**Proof.** This follows from Theorem 1, Theorem 2 and (VK-Square Theorem - to be added). \( \Box \)

This result allows DPO graph rewriting to be applied in our category \( \mathbb{GL} \).

**Definition 13** \((\text{double pushout rewriting})\)

A graph production \( L \xrightarrow{2} K \xrightarrow{1} R \) is applied to a host graph \( G \) with the following procedure:
First, find a morphism $m$ that maps $L$ to $G$.

Then, find a morphism $k$ that maps $K$ to $D$ such that: the pushout of $K \xrightarrow{l} L$ and $K \xrightarrow{k} D$ is $G$ (with $m$).

Then, build the pushout of $K \xrightarrow{l} R$ and $K \xrightarrow{k} D$, which is the result of applying the rule.

Note furthermore that:

- If either of the morphisms $m$ or $k$ does not exist, the rule cannot be applied.
- In an adhesive HLR category, when $l$ and $r$ are both monomorphisms from the subclass $M$, morphism $k$ is unique (if it exists). This ensures that the result of applying the rule is determined completely (up to isomorphism) by $m$.

Unfortunately, the current definitions, although sound, still give rise to some strange behaviour. Suppose that $p = (L \leftarrow K \to R)$ is a production. Then:

- If $R$ contains list nodes that have no counterpart in $K$, then the application of $p$ introduces list nodes in the host graph. This is undesirable, because a list node in a normal graph has no meaning; a list node only makes sense in a rule.

- Conversely, if $L$ contains list nodes that have no counterpart in $K$, then $p$ can never be applied to graphs that do not contain list nodes. This is due to the pushout construction (see Theorem 3), which copies $(B \setminus A)$ (in this case $(L \setminus K)$ into the pushout (in this case the intermediate host graph $D$).

We will disallow this strange behaviour by demanding that both the morphisms in a production must be surjective with respect to list nodes, which ensures that $L$ and $R$ cannot contain list nodes that do not have a counterpart in $K$.

**Definition 14 (surjective $M$-morphisms)**

A $M$-morphism $f = (f_V, f_E) : G \to H$ in $\mathcal{GL}$ is list surjective if for all $v_H \in V_H$ there exists a $v_G \in V_G$ such that $f_V(v_G) = \langle v_H \rangle$.

**Definition 15 (productions in $\mathcal{GL}$)**

For graph rewriting in the category $\mathcal{GL}$, only productions $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ are allowed in which both $l$ and $r$ are list surjective $M$-morphisms.

It turns out that $l$ and $r$ being surjective is not only a necessary, but even a sufficient condition for ensuring that rules do not introduce list nodes. This is formalised in Theorems 4 and 5. Consequently, graph rewriting in our category $\mathcal{GL}$ of list graphs is indeed sound.

**Definition 16 (list count)**

The list count of a graph $G = (\hat{V}, V, E, src, tgt, lab)$ is the size of $V$ and is denoted by $LC(G)$.

**Theorem 4 (valid rule match implies that $l$ is surjective)**

In $\mathcal{GL}$, if a production $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ matches on a graph $G$ with $LC(G) = 0$, then $l$ must be list surjective.

**Proof.**
Assume that the application of \( p \) on \( G \) takes place according to the following DPO diagram:

\[
\begin{array}{ccc}
L & \xleftarrow{l} & K & \xrightarrow{r} & R \\
\downarrow m & & \downarrow i & & \downarrow m' \\
G & \xleftarrow{l'} & I & \xrightarrow{r'} & H
\end{array}
\]

Without loss of generality, we may assume that \( I \) and \( L \) are disjoint, and that \( G \) is built with the pushout construction of Definition 12. This means that \( LC(G) = |V_G| = |V_I \cup V_L \setminus I(l)| = |V_I| + |V_L \setminus I(l)| = 0 \) holds, and therefore \( |V_L \setminus I(l)| = 0 \). Therefore, \( V_L \subseteq I(l) \) must hold, which implies that \( l \) is list surjective.

\[\square\]

**Theorem 5 (surjective rules do not introduce list nodes)**

In \( \mathcal{GL} \), if a production \( p = (L \leftarrow l \rightarrow K \rightarrow r \rightarrow R) \) with \( l \) and \( r \) list surjective is applied on a graph \( G \) leading to result \( H \), and \( LC(G) = 0 \), then \( LC(H) = 0 \) as well.

**Proof.**

Assume that the application of \( p \) on \( G \) takes place according to the following DPO diagram:

\[
\begin{array}{ccc}
L & \xleftarrow{l} & K & \xrightarrow{r} & R \\
\downarrow m & & \downarrow i & & \downarrow m' \\
G & \xleftarrow{l'} & I & \xrightarrow{r'} & H
\end{array}
\]

Without loss of generality, we may assume that \( I \) and \( L \) are disjoint, and that \( G \) is built with the pushout construction of Definition 12. This means that \( LC(G) = |V_G| = |V_I \cup V_L \setminus I(l)| = |V_I| + |V_L \setminus I(l)| = 0 \) holds, and therefore \( |V_I| = 0 \).

Also, without loss of generality, we may assume that \( I \) and \( R \) are disjoint, and that \( H \) is built with the pushout construction of Definition 12. This means that \( LC(H) = |V_H| = |V_I \cup V_R \setminus I(r)| = |V_I| + |V_R \setminus I(r)| = |V_R \setminus I(r)| \) holds. Because \( r \) is list surjective, \( V_R \subseteq I(r) \), therefore \( |V_R \setminus I(r)| = 0 = LC(H) \).

\[\square\]

4 List reversal

We show some more applications of list graph transformations, inspired by the setting of Section 2. In particular, we show how we can obtain a participants list, \( \text{parts} \), from a ranking list, \( \text{rank} \), by copying and reversing the list. The entire behaviour is specified by the three rules in Figure 6.

- The start rule copies the rank list into a copy list, and creates an empty parts list. Note that this is a “shallow” copy: the elements are not copied but shared among the lists.

- The build rule repeatedly removes the last element from the copy list and appends it to the parts list. By applying this rule as long as possible, eventually the copy list will be empty, at which point the parts list contains all the elements of the original copy list, and hence of the rank list, in reverse order.
Figure 6: List graph rules creating a reversed parts list out of a rank list.

Figure 7: Example production sequence for the rules in Figure 6.

- The finish rule deletes the empty copy list, completing the reversal process. Note that this rule is only applicable if the copy list is indeed empty.

Figure 7 shows a sequence of applications of these rules.

5 Conclusion

In this section, we look back on what we have achieved, and list the good and bad points. We also briefly discuss related work and future extensions.

5.1 Evaluation

We have defined list graphs in order to directly capture ordered structures. We have shown that encoding such structures into plain graphs is awkward and, worse, introduces programming-like
structures that break the inherent abstraction of graph-based models. In contrast, the construction and manipulation of list graphs is much more abstract and results in smaller, more intuitive graphs and rules. We have also shown that list graphs fit into the theory of algebraic graph rewriting, and so the cost of the more complex graph formalism is low, at least on the level of theory.

On the downside, the way lists are manipulated on the theoretical level is not attractive from an implementation point of view. List edges are deleted and created as a whole, which, when taken literally, would mean that entire lists are discarded and constructed every time a single element is added or deleted. An implementation should instead recognise and efficiently deal with frequently occurring patterns of list usage. A first attempt is to identify re-use of list edges with a static analysis of stable nodes and edges, but it is yet unclear how this can be generalised.

It may be remarked that our list edges break the usual symmetrical treatment of edge sources and targets, since lists may only occur at the target and not at the source of an edge. In this regard, we have been led by the intended application of the enriched formalism. From the theoretical perspective there is no reason to forbid list nodes at edge sources: our theory smoothly extends to standard hyperedges (keeping our specialised notion of morphism), which do not have a distinguished source node at all.

5.2 Related work

As far as we have been able to determine, there is essentially no prior work on enriching the basic graph formalism with lists. On a more pragmatic level, however, many tools offer ways to deal with ordered structures or associations, if only by suggesting a default encoding or syntactic sugar. For instance, FUJABA reflects programming structures such as lists and arrays into the rules, and provide notations to traverse them conveniently (see [MZ04]). FUJABA’s handling of ordered edges is formalised in [Zün01]. For VIATRA2 it is suggested in [VB07] to use relations over relations to encode ordering. In general it is difficult to find information about such pragmatic solutions.

Remotely related are extensions to deal with parallel or amalgamated rule applications (e.g., [Tae97]), since in this setting the rules also have nodes that can be mapped to more than one graph node (a prime instance are the set nodes of PROGRES, see [Sch97]). However, the connection stops there: the purpose and technical contribution of this work is entirely different.
5.3 Future work

So far, the concepts in this paper only exist in theory. The proof of their usability can only come through an implementation. The natural way to go is to extend our research vehicle GROOVE (see [Ren04]) to list graphs. However, this will require a major refactoring to generalise to hyperedges — quite apart from the fact that GROOVE implements SPO and not DPO rewriting.

Instead, we first plan to use these ideas to define a suitable transformation language in the project CHARTER\(^1\), in the context of which this work has been carried out. For this project we will provide a tool that compiles graph transformation systems to Java source code which accesses and manipulates the actual graphs through a predefined API. Since ordered lists and arrays are a common feature in the graphs we will have to deal with, it is imperative to have a suitable, declarative way to specify their transformation.

A theoretical extension that would add quite a bit of power to the formalism, and make it even more generally usable, is indexing. Currently there is no way to specify or reason about the position of an element in a list. We conjecture that this requires only a minor extension, namely to add a default unmodifiable length attribute to all list nodes. Morphisms then have to respect the length of list nodes, in the following way: if a morphism maps a list node to another list node, then the value of the length attribute should remain unchanged, whereas if the image is a sequence of plain nodes, the value of the length attribute should equal the actual length of the sequence. For instance, Figure 8 specifies that a Part-node should be inserted at index \(i\).

Bibliography


\(^1\) See [http://charterproject.ning.com/](http://charterproject.ning.com/).


