Knowledge representation and update in hierarchies of graphs

Russ Harmer, Eugenia Oshurko

▶ To cite this version:

Knowledge representation and update in hierarchies of graphs

Russ Harmer, Eugenia Oshurko

Univ Lyon, EnsL, UCBL, CNRS, LIP, F-69342 LYON Cedex 07, France

Abstract

A mathematical theory is presented for the representation of knowledge in the form of a directed acyclic hierarchy of objects in a category where all paths between any given pair of objects are required to be equal. The conditions under which knowledge update, in the form of the sesqui-pushout rewriting of an object in a hierarchy, can be propagated to the rest of the hierarchy, in order to maintain all required path equalities, are analysed: some rewrites must be propagated forwards, in the direction of the arrows, while others must be propagated backwards, against the direction of the arrows, and, depending on the precise form of the hierarchy, certain composability conditions may also be necessary. The implementation of this theory, in the `ReGraph` Python library for (simple) directed graphs with attributes on nodes and edges, is then discussed in the context of two significant use cases.

Keywords: knowledge representation, graph rewriting, graph databases

1. Introduction

We present a generic framework for knowledge representation (KR) based on hierarchies of objects from an appropriately structured category: a hierarchy is a directed acyclic graph (DAG) whose nodes are *objects* of the category and whose edges are *arrows* of the category such that all paths between each pair of objects are equal; we refer to this as the *commutativity* condition.

The principal model of interest to us in this paper uses (simple) graphs and homomorphisms so that a hierarchy is a DAG whose nodes are themselves (simple) graphs. In this model, an edge of the DAG $h : G \to T$ asserts that the graph G is *typed* by T, i.e. T defines the kinds of nodes and kinds of edges (and attributes, if desired) that exist in G and h specifies, for each node and edge (and attribute) of G, which kind it is. As such, T can be viewed as a more abstract representation of knowledge of which G provides a more concrete instantiation.

We require certain structure on the category in order to be able to perform sesqui-pushout rewriting [4] to update an object in the hierarchy. However, such an update may invalidate some of the typing arrows of the hierarchy. The main contribution of this paper is to present a mathematical theory that guarantees the reconstruction of a valid hierarchy, after an arbitrary rewrite of an object, by appropriately *propagating* that rewrite to the other objects in the hierarchy. In general, this only concerns a subgraph of the hierarchy that is determined as a function of the nature of the update and the paths to and/or from the updated object. In the case where there are multiple paths between a given pair of objects of the hierarchy, this reconstruction depends on the satisfaction of a *composability* condition, guaranteeing that the propagated rewrites are compatible, in order to maintain validity of the commutativity condition.

Graph databases

Modern database systems are increasingly migrating towards graph-based representations as a response to the growing wealth of data—from domains as varied as social or transport networks, the semantic web or biological interaction networks—that are most naturally expressed in those terms. However, unlike traditional relational DBs or earlier graph-based formats such as RDF, most graph DBs based on the richer model of property graphs [9, 2] do not provide a native notion of *schema*. Our notion of hierarchy provides a mathematical framework for this. Indeed, an explicitly given schema graph to which a data, or instance, graph is homomorphic is the simplest non-trivial example of a hierarchy in our sense: the nodes of the schema specify the types of entites allowed in the system; its edges specify which edges between different types of nodes are allowed; and the attributes on its nodes and edges define the set of permitted attributes for nodes and edges. As such, the existence of a homomorphism from a data graph to a schema graph provides a proof of schema *validation* [2].

Our theory of propagation of rewriting in a hierarchy precisely captures the ways in which schema-aware DBs can be updated: a *descriptive* update occurs when the data is modified and the schema has to adjust accordingly; while a *prescriptive* update occurs when the schema is modified and the data needs to be adjusted [2]. More precisely, if we *add* a node to the data graph and choose not to specify that its type already exists in the schema graph, in order to maintain the homomorphism from data to schema, we must propagate this operation to the schema graph to create a new node in the schema graph to type the new node of the data graph; similarly, if we *merge* two nodes of different types of the data graph, we must merge the corresponding typing nodes of the schema. Conversely, if we *delete* a node of the schema graph, we can only maintain the homomorphism by deleting all instances of that node in the data graph; and if we *clone* a node of the schema and choose not specify how to retype its instances in the data graph, those instances must be cloned in the data graph. In summary, *add* and *merge* updates propagate *forwards*, in the direction of the typing homomorphism, while *clone* and *delete* updates propagate *backwards*; and, as we will show, these observations remain true for general hierarchies.

Our theory thus provides a specification of how to enforce an *abstraction barrier* on a schema-less graph DB that provides the illusion of being schema-aware. Our Python library `ReGraph` implements this for the Neo4j graph DB by fixing an encoding of the data and schema graphs and the typing homomorphism within the single graph provided by Neo4j and translating any combination of clone, delete, add and merge operations into a corresponding query written in the Cypher query language used by Neo4j [2]. More importantly, our theory also provides a specification of how to enforce the abstraction barrier for an *arbitrary* hierarchy—modulo the need to fix the encoding into Neo4j and the translation of update operations into Cypher. However, these two requirements are generic and can be derived systematically. As such, we provide the foundations for exploiting Neo4j (or similar graph DBs) as a platform for arbitrary, user-defined graph-based KR systems.

The KAMI bio-curation tool

The core of the `KAMI` bio-curation system [13] has a richer 3-level hierarchy. At the root lies its *meta-model*, a fixed, hard-wired graph which defines the universe of discourse pertinent to the rule-based modelling of protein-protein interactions (PPIs) in cellular signalling: genes, regions of genes, binding and enzymatic actions, &c. The meta-model types an *action graph* which defines the particular collection of genes (and so on) of interest to a *corpus* of knowledge, e.g. a signalling pathway. The action graph types a collection of *nugget* graphs, each representing the detailed conditions needed for a particular PPI to occur. In other words, an action graph summarizes the *anatomy* of a system while the collection of nugget graphs provides a representation of the *physiology* that determines how the system can behave.

In general, an update of a nugget graph refers to some anatomic features that already exist in the action graph and to others that must be added to maintain typing; this is performed automatically by forward propagation. It is important that propagation does not continue to the meta-model (which must remain unchanged); this is achieved by requiring that all new anatomic features specify (at least) how they are to be typed by the meta-model. This gives an example of the notion of *controlled* forward propagation, as discussed in section 3, and can be seen as a more general instance of a descriptive DB update which actually preserves the current schema.

A knowledge corpus in `KAMI` can be contextualized, with respect to a choice of gene products, through an update of its action graph, giving rise to what we call a `KAMI` *model*; in the terminology of DBs, this is analogous to a *materialized view*—a contextualized copy of part of the original DB that can be manipulated independently. The effect of this update propagates backwards to the nugget graphs. This propagation is not controlled—the cloning of a gene precisely gives rise to multiple gene products—unlike the case of *concept refinement* where the cloning of a schema node is accompanied by a specification of how to retype all instances of the original node in the data graph in terms of the refined schema. We discuss backward propagation, including the controlled case, in section 4.

Related work

Slice categories provide many rich models of *typed* sesqui-pushout rewriting [4], e.g. **Set**$/T$ defines a setting for multi-set rewriting over the set T. We provide a powerful generalization of this where, through the use of a hierarchy, we can not only guarantee that rewriting an object always returns a well-typed result but, additionally, can dynamically modify the typing object T. Our approach is related to the change-of-base functor familiar from algebraic topology and to its right adjoint whose existence characterizes pullback complements [7]. Indeed, in a sense, our work can be seen as providing a means of exploiting this theory, in a form that can be used for knowledge representation and graph databases, even when only those PBCs required for SqPO rewriting exist.

The arrows in our hierarchies correspond intuitively to the type, or instance-of, relationships found in entity-relationship (ER) modelling [3] or UML, i.e. they are relations that cross from one meta-model layer to another. They also generally correspond to TBox statements in Description Logic [1] although, in some cases, this intuition breaks down since an object, such as the nugget graph of KAMI, with no incoming arrows usually corresponds to a collection of ABox statements about instances of the concepts defined below it in the hierarchy. In this paper, we do not consider the specialization/generalization, or is-a, relationships found in ER modelling for the reason that the rewrite of an object does not need to propagate across such relations.

2. Preliminaries

In this section, we discuss the necessary preliminary material concerning graph rewriting—specifically the definitions of pullback complements and image factorizations—and provide a formal definition of our notion of hierarchy.

Let us begin by defining a piece of useful terminology. We use the term *element* to refer to any concrete constituent of an object in a concrete category of interest to us, e.g. an element (in the usual sense) of a set or a node, edge or attribute of a graph.

2.1. Sesqui-pushout rewriting

Sesqui-pushout (SqPO) rewriting [4] is a generalization of double pushout (DPO) and single pushout (SPO) rewriting [5, 8]. In typical concrete settings, it allows for the expression of rules for all elementary manipulations generally considered in traditional graph (or multi-set) rewriting: the addition, deletion, merging and cloning of elements as well as the modification of the value(s) associated with an attribute. It extends SPO rewriting, by allowing for cloning, which in turn extends DPO rewriting by allowing for side-effects due to deletion (but not those due to merging, which DPO rewriting already accommodates).

The abstract formulation of SqPO rewriting requires the categorical notion of final pullback complements (PBCs) [7]. As this remains (slightly) non-standard, we include a full definition here.

Given a pair of composable arrows $f : A \to B$ and $g : B \to D$, their *final pullback complement* is a pair of composable arrows $\hat{g} : A \to C$ and $\hat{f} : C \to D$ such that the resulting square is a pullback (PB) satisfying the following universal property (UP): given a PB square (using g but not necessarily f)

$$
\begin{array}{ccc}
B & \xleftarrow{\;f'\;} & A' \\
{\scriptstyle g}\big\downarrow & & \big\downarrow{\scriptstyle \hat{g}'} \\
D & \xleftarrow{\;\hat{f}'\;} & C'
\end{array}
$$

and an arrow $h : A' \to A$ such that $f' = f \circ h$, there exists a unique arrow $\hat{h} : C' \to C$ such that $\hat{f}' = \hat{f} \circ \hat{h}$ and $\hat{g} \circ h = \hat{h} \circ \hat{g}'$.

SqPO rewriting can be performed in any category with all PBs, all PBCs over monos, i.e. where $g : B \to D$ is a mono, and all pushouts (POs); we further require that POs preserve monos. These conditions are satisfied in all concrete settings of interest to us, typically sets and (simple) graphs with attributes, and potentially in many other concrete settings to which our theory would therefore also apply.

In order to perform SqPO rewriting of a single object, we only actually need the existence of PBs and POs of (co-)spans where one arrow is a mono. However, in this paper, we sometimes have need of more general PBs and POs to express the propagation of rewriting through a hierarchy. We also need the existence of all *image factorizations* (IFs). As this notion is not standard in graph rewriting, we give an explicit definition of its UP.

The image factorization of an arrow $f : A \to B$ is a mono $m : I \rightarrowtail B$ such that (i) there exists an arrow $e : A \to I$ such that $f = m \circ e$; and (ii) for any arrow $e' : A \to I'$ and mono $m' : I' \rightarrowtail B$ such that $f = m' \circ e'$, there exists a unique arrow $i : I \to I'$ such that $m = m' \circ i$.

In the concrete settings of interest to us, the IF of an arrow coincides with the familiar notion of its epi-mono factorization. However, we have no (abstract) need for the first arrow to be an epi and so prefer the more abstract requirement of having IFs of all arrows.

We consider a *rule* to be simply an arrow. A *restrictive instance* of a rule $r^- : L \leftarrow P$ in an object G is a mono $m : L \rightarrowtail G$ from the target object L; in this case, we refer to L as the LHS and P as the RHS of r^-. An *expansive instance* of a rule $r^+ : P \rightarrow R$ is a mono from the source object P; in this case, we refer to P as the LHS and R as the RHS of r^+.

The usual notion of rule, i.e. a span of arrows, consists of two rules, r^- and r^+, in our sense with a common source object P. Given a restrictive instance m of the first, the PBC of r^- and m provides an expansive instance m^- of the second and the PO of m^- and r^+ completes the overall rewrite of G to G^+.

$$
\begin{array}{ccccc}
L & \xleftarrow{\;r^-\;} & P & \xrightarrow{\;r^+\;} & R \\
{\scriptstyle m}\big\downarrow & & \big\downarrow{\scriptstyle m^-} & & \big\downarrow{\scriptstyle m^+} \\
G & \xleftarrow[\;g^-\;]{} & G^- & \xrightarrow[\;g^+\;]{} & G^+
\end{array}
$$

2.2. Hierarchies

We formalize the notion of hierarchy by first defining the underlying skeleton of the KR as a DAG then defining the hierarchy itself as a graph homomorphic to the skeleton. More precisely, a *skeleton* is a finite directed acyclic simple graph and a *hierarchy* over the skeleton $\mathcal{S}$ is a finite directed simple graph $\mathcal{H}$ equipped with a homomorphism to $\mathcal{S}$; as such, it is also directed acyclic.

The simplest non-trivial skeleton consists of two nodes, d and s, with a single edge $e : d \rightarrow s$ between them; this expresses that there are two types of object—let us call them *data* and *schema* nodes—and that data nodes have edges to schema nodes. The simplest hierarchy over this skeleton is the skeleton itself; this defines a KR system that contains a single data node, a single schema node and an edge from the former to the latter. However, a hierarchy over this skeleton could contain multiple data and/or schema nodes where a single data node may have edges to multiple schema nodes and/or multiple data nodes may have edges to a single schema node.

More generally, a skeleton specifies the *types* of nodes and edges that can exist, i.e. the *shape* of the KR, while a hierarchy specifies the *instances* of those objects and arrows that actually exist, i.e. the current *structure* of the KR. The skeleton of a KR system generally remains invariant throughout its lifetime while its hierarchy evolves over time. However, in this paper, we do not consider operations that modify the structure of a hierarchy; instead, we are interested in instantiating hierarchies with content and in modifying that content.

The *instantiation* of a hierarchy $\mathcal{H}$ in a category $\mathbf{C}$ is defined by assigning an object $[\![n]\!]$ of $\mathbf{C}$ to each node n of the hierarchy and an arrow $[\![e]\!] : [\![n_1]\!] \rightarrow [\![n_2]\!]$ of $\mathbf{C}$ to each edge $e : n_1 \rightarrow n_2$ of the hierarchy in such a way that the commutativity condition is satisfied. (This can be seen as a functor from the reflexive, transitive closure of $\mathcal{H}$—defined in such a way as to be still a simple graph—to $\mathbf{C}$.)

An instantiation of the above hierarchy $e : d \to s$ in **Set** therefore consists of two sets, G and T, and a function $h : G \to T$ between them; this can be seen as an intensional representation of a multi-set where G defines the individuals, T defines the types of individuals and h assigns a type to each individual. We are interested in operations that update (one or other of) these two sets, e.g. adding an element to G or removing an element from T. However, such operations necessitate the making of changes to the function h and, potentially, to the other set as well, e.g. if we add an element to G, we must update h to be defined on that new element—and this may entail adding a new element to T, if it does not already contain the desired type.

Our theory provides a general framework for expressing and applying such updates of objects, as SqPO rules, and determines how the arrows and other objects must be updated in consequence in order to maintain a valid instance of the hierarchy. In the next two sections, we explain how (i) an expansive rewrite of G is *propagated* to T in order to obtain a rewritten hierarchy $h^+ : G^+ \to T^+$; and (ii) a restrictive rewrite of T is propagated to G in order to obtain a rewritten $h^- : G^- \to T^-$.

3. Forward propagation

Throughout this section and the next, we consider two objects G and T and an arrow $h : G \to T$ of a category $\mathbf{C}$ possessing all the structure required for SqPO rewriting, i.e. an instantiation in $\mathbf{C}$ of the hierarchy $e : d \to s$.

In this section, we consider a rule $r : L \to L^+$ and an expansive instance $m : L \rightarrowtail G$ of r in G. Note that we immediately obtain a typing of L by T by composition, i.e. $h \circ m : L \to T$.

3.1. The strict phase of forward rewriting

In order to decide how to propagate a rewrite of G to T, we must further specify to what extent we wish to consider the RHS L^+ of r to be typed by T. There are two extreme cases: the first is where we provide an arrow from L^+ to T, i.e. L^+ is itself typed by T; the other is the case where nothing in the complement of the image of r is homomorphic to T. In the first case, which we call a *strict* rewrite of G, the rewritten G^+ is still typed by T; in the other case, which we call the *canonical* propagation to T, we must propagate all changes in G to T. In between these extremes, we can specify those elements, not in the image of r, that we nonetheless wish to be typed by T.

Definition. Given a rule $r : L \to L^+$, a *forward factorization* of r is an object L' and arrows $r' : L \to L'$ and $r^+ : L' \to L^+$ such that $r = r^+ \circ r'$; and an arrow $x : L' \to T$ such that $h \circ m = x \circ r'$.

$$
\begin{array}{ccc}
L & \xrightarrow{\ r\ } & L^+ \\
{\scriptstyle h \circ m}\big\downarrow & \searrow{\scriptstyle r'} & \big\uparrow{\scriptstyle r^+} \\
T & \xleftarrow{\ x\ } & L'
\end{array}
\tag{1}
$$

7

In the case of strict rewriting, L' is isomorphic to L^+ so that $x : L^+ \to T$ whereas, if L' is isomorphic to L, x specifies nothing more than $h \circ m$. In the concrete settings of multi-sets and of graphs, r' is frequently taken to be a mono, i.e. it expresses a rule that only *adds* elements that can be typed by T, but in the abstract setting we have no need to enforce this as a requirement.

The factorization of r splits its application into two phases: the *strict* phase, specified by r', which modifies only G; and the *canonical* phase, specified by r^+, which modifies G and T.

Definition. The *strict rewrite* of G is defined by taking the PO of m and r'. By the definition (1) of forward factorization and the universal property of this PO, we obtain a (unique) arrow h' that types G' by T. Note that $x = h' \circ m'$.

$$
\begin{array}{ccc}
L & \xrightarrow{r'} & L' \\
{\scriptstyle m}\downarrow & & \downarrow{\scriptstyle m'} \\
G & \xrightarrow{g'} & G' \quad\Big)\,x \\
& {\scriptstyle h}\searrow & \downarrow{\scriptstyle h'} \\
& & T
\end{array}
\tag{2}
$$

Note that the strict rewrite can only merge elements of G that have the same type; this is a consequence of the requirement that $h \circ m = x \circ r'$. It can also add multiple elements to G—provided they can all be typed in T.

This strict phase of rewriting was discussed briefly in [10] as being the only kind of rewrite that can be performed if T is hard-wired as the base object of a slice category; typically, a descriptive update that *preserves* the current schema.

3.2. The canonical phase of forward propagation

Our more general and flexible setting of hierarchies enables a second phase of rewriting where the remaining changes to be made to G', as specified by r^+, are additionally propagated to T, i.e. the base object changes.

Definition. The *rewrite* of G is completed by taking the PO of r^+ and m'. The *forward propagation* to T is then defined by taking the PO of g^+ and h'. The final typing of G^+ by T^+ is given by h^+.

$$
\begin{array}{ccccccc}
L' & \xrightarrow{r^+} & L^+ & \qquad & G' & \xrightarrow{g^+} & G^+ \\
{\scriptstyle m'}\downarrow & & \downarrow{\scriptstyle m^+} & & {\scriptstyle h'}\downarrow & & \downarrow{\scriptstyle h^+} \\
G' & \xrightarrow[g^+]{} & G^+ & & T & \xrightarrow[t^+]{} & T^+
\end{array}
\tag{3}
$$

Note that, by the pasting lemma for POs, since the phased rewrite of G, by r' then r^+, occurs through consecutive POs, its overall effect is the same as that obtained by applying r directly. Note also that we could instead have constructed T^+ by taking the PO of r^+ and $x = h' \circ m'$ and applying the UP of G^+ to construct h^+; the two approaches are equivalent by pasting for POs.

8

The propagated rewrite $t^+ : T \to T^+$ performs all the additions and merges, as specified by r^+ for G', in T to produce the new type T^+ required for G^+. We can alternatively obtain this rewrite by projecting r^+ to a new rule that applies specifically to T.

Definition. The *projection* $\hat{r}^+ : L_T \to L_T^+$ of r^+ to T is computed by taking the IF of $h' \circ m'$ followed by the PO of r^+ and $\hat{h}'$. It immediately has an expansive instance $\hat{m}'$ in T and we obtain T^+ by taking the PO of $\hat{m}'$ and $\hat{r}^+$.

$$
\begin{array}{ccc}
L' & \xrightarrow{\ r^+\ } & L^+ \\
& \searrow^{\hat{h}'} & \\
h'\circ m' & L_T \xrightarrow{\ \hat{r}^+\ } L_T^+ & \\
& \downarrow^{\hat{m}'} \qquad \downarrow^{\hat{m}^+} & \\
& T \xrightarrow{\ t^+\ } T^+ &
\end{array}
\tag{4}
$$

The retyping of G^+ by T^+ follows from a straightforward application of the UP of the PO, in (3), defining G^+.

It is easy to show, by the pasting lemma for POs, that these two definitions of T^+ coincide; as such, for the instance $h : G \to T$ of the simple hierarchy $e : d \to s$, we can use either. However, in a more general setting where T may be typed by further objects, we must compute the rule projection explicitly in order to continue propagation; we return to this in section 5.

3.3. The forward clean-up phase

The strict phase of rewriting allows us to add elements to G that can already be typed by T. However, if we wish to add elements that cannot be typed by T, this must occur during the canonical phase of rewriting; as such, every such element acquires a *distinct* type in the updated T^+.

In order to allow the addition of multiple elements of the *same* new type in T^+, we allow the specification of a *clean-up* phase of rewriting, that applies only to T^+ (and not G^+), by providing an epi $r^\oplus : L_T^+ \twoheadrightarrow L_T^\oplus$; this allows us in particular to merge two newly-added elements of T^+. However, this requires us to know L_T^+—which is dependent on the typing $h' : G' \to T$ and so cannot be specified statically, at the same time as r, but rather dynamically when r's rewrite is propagated to T.

Definition. The clean-up phase is specified by an epi $r^\oplus : L_T^+ \twoheadrightarrow L_T^\oplus$ and the expansive instance $\hat{m}^+ : L_T^+ \rightarrowtail T^+$, obtained after the rewrite of T with the rule projection $\hat{r}^+$ above, giving rise to the final retyping $t^\oplus \circ h^+ : G^+ \to T^\oplus$ of G^+.

$$
\begin{array}{ccc}
G^+ & L_T^+ \xrightarrow{\ r^\oplus\ } L_T^\oplus & \\
& \downarrow^{\hat{m}^+} \qquad \downarrow^{\hat{m}^\oplus} & \\
h^+ \searrow & T^+ \xrightarrow{\ t^\oplus\ } T^\oplus &
\end{array}
\tag{5}
$$

9

Let us note that if r' adds an element e_1 to G and r^+ adds a second e_2, the clean-up phase may merge the newly-added element of T^+ with the element of T that types e_1, so that e_1 and e_2 have the same type in $T^\oplus$. This enables us to correct the update in the case where, at the time of defining r and its factorization, we failed to realize that e_2 could actually be typed in T.

We ask for $r^\oplus$ to be an epi because, in all concrete models of interest to us, this corresponds to a rule that *only* merges nodes of T^+ in the image of $\hat{m}^+$ and we have no use case for using clean-up to add new elements to T^+.

3.4. Example

Let us illustrate the above theory in a case where G and T are sets, i.e. the hierarchy represents a multi-set. The object T has two elements—white and black circle—and G has two instances of each (we use this colour coding to avoid specifying the homomorphisms explicitly). The rule specifies (i) the *merge* of one white and one black circle; and (ii) the addition of two squares (which we wish to have the same type).
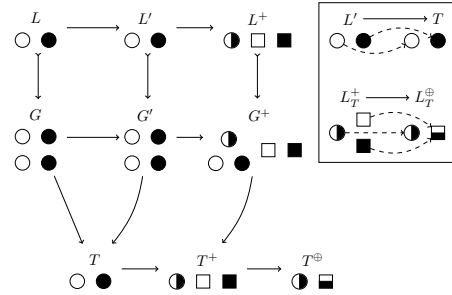
The strict phase can neither merge the circles nor add the squares; as such, everything must occur in the canonical phase which performs, and propagates, the merge and additions. Note that this has the *side-effect* that the two circles of G not directly concerned by the rewrite have nonetheless been retyped in T^+. The clean-up phase now allows us to merge the two newly-added squares of T^+ so that the two squares in G^+ have the same type in $T^\oplus$.

If a square already exists in T, we could factorize the rule differently to add one square in the strict phase. In this case, clean-up can be used to merge the single newly-added square in T^+ with the one that existed in T; the overall effect is the same as if we had simply added both squares in the strict phase.

10
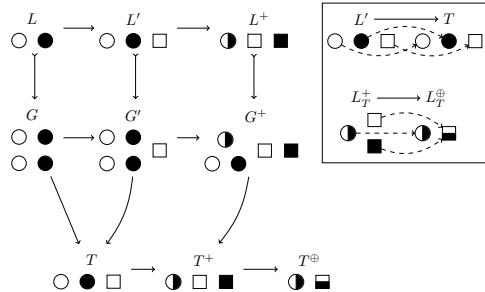
4. Backward propagation

In this section, we consider a rule $r : L \leftarrow L^-$ with a restrictive instance $m : L \rightarrowtail T$ in T. We can immediately compute the PB of h and m to obtain a span $\hat{m} : G \leftarrowtail L_G \rightarrow L : \hat{h}$ from the object L_G that can be seen as the sub-object of G whose *typing* by T can be modified by r.

$$
\begin{array}{ccc}
L_G & \xrightarrow{\hat{h}} & L \\
\hat{m}\downarrow & & \downarrow m \\
G & \xrightarrow{h} & T
\end{array}
\tag{6}
$$

4.1. The strict phase of backward rewriting

Analogously to forward propagation, we must provide a factorization of r in order to specify which changes to T are to be propagated to G.

Definition. Given a rule $r : L \leftarrow L^-$, a *backward factorization* of r is an object L' and arrows $r' : L \leftarrow L'$ and $r^- : L' \leftarrow L^-$ such that $r = r' \circ r^-$; and an arrow $\hat{h}' : L_G \to L'$ such that $\hat{h} = r' \circ \hat{h}'$. Note that L_G (not G) plays the role analogous to T in forward propagation.

$$
\begin{array}{ccc}
L & \xleftarrow{r} & L^- \\
\hat{h}\uparrow & \nwarrow_{r'} & \downarrow r^- \\
L_G & \xrightarrow{\hat{h}'} & L'
\end{array}
\tag{7}
$$

The factorization of r splits its application into two phases: the *strict* phase, specified by r', which modifies only T; and the *canonical* phase, specified by r^-, which modifies G and T. As such, in the strict phase of restrictive rewriting, G and L_G remain invariant.

Definition. The *strict rewrite* of T is defined by taking the PBC of r' and m. By definition (7) and an application of the UP of this PBC to the PB, in (6), defining L_G, we obtain the retyping of G as $h' : G \to T'$.

Note that any element of T that is deleted must have *no instances* in G for this to be possible—this is a consequence of the requirement that $\hat{h} = r' \circ \hat{h}'$; and that, if an element of T is cloned, *all* its instances in G are reassigned a *unique* type in T' by $\hat{h}'$, i.e. we are performing a *concept refinement*.

Note also that, by the inverse pasting lemma for PBs, the resulting square is itself a PB:

$$
\begin{array}{ccc}
L_G & \xrightarrow{\;\hat{h}'\;} & L' \\
{\scriptstyle \hat{m}}\downarrow & & \downarrow{\scriptstyle m'} \\
G & \xrightarrow[\;h'\;]{} & T'
\end{array}
\tag{8}
$$

4.2. The canonical phase of backward propagation

Definition. The *rewrite* of T is completed by taking the PBC of r^- and m'. The *backward propagation* to G is then defined by taking the PB of h' and t^-. The final typing of G^- by T^- is simply h^-.

$$
\begin{array}{ccc}
L' & \xleftarrow{\;r^-\;} & L^- \\
{\scriptstyle m'}\downarrow & & \downarrow{\scriptstyle m^-} \\
T' & \xleftarrow[\;t^-\;]{} & T^-
\end{array}
\qquad\qquad
\begin{array}{ccc}
G & \xleftarrow{\;g^-\;} & G^- \\
{\scriptstyle h'}\downarrow & & \downarrow{\scriptstyle h^-} \\
T' & \xleftarrow[\;t^-\;]{} & T^-
\end{array}
\tag{9}
$$

This construction is analogous to the direct construction of T^+ as a PO in forward propagation. If the strict phase of rewriting is trivial, i.e. $L \cong L'$, this corresponds exactly to the notion of (backward) propagation defined in [10]. Note that, by the horizontal pasting lemma for PBCs (see, for example, Proposition 5 of [14]), the overall effect of r' followed by r^- on T is the same as that obtained by applying r directly.

The propagated rewrite $g^- : G \leftarrow G^-$ performs all the clones and deletions, as specified by r^- for T', in G to produce the new object G^- typed by T^-. We can also obtain this by constructing a new rule and applying it directly to G.

Definition. The *lifting* $\hat{r}^- : L_G \leftarrow L_G^-$ of r^- to G is computed by taking the PB of $\hat{h}'$ and r^-. It immediately has the restrictive instance $\hat{m}$, from (6), in G from which we obtain G^- by taking the PBC of $\hat{r}^-$ and $\hat{m}$.

$$
\begin{array}{ccc}
L_G & \xleftarrow{\;\hat{r}^-\;} & L_G^- \\
{\scriptstyle \hat{h}'}\downarrow & & \downarrow{\scriptstyle \hat{h}^-} \\
L' & \xleftarrow[\;r^-\;]{} & L^-
\end{array}
\qquad\qquad
\begin{array}{ccc}
L_G & \xleftarrow{\;\hat{r}^-\;} & L_G^- \\
{\scriptstyle \hat{m}}\downarrow & & \downarrow{\scriptstyle \hat{m}^-} \\
G & \xleftarrow[\;g^-\;]{} & G^-
\end{array}
\tag{10}
$$

In this case, we must construct the new typing of G^- by T^- by applying the pasting lemma for PBs and the UP of the PBC defining T^-, in (9), to obtain $h^- : G^- \to T^-$.

$$
\begin{array}{ccccc}
L' & \xleftarrow{\;\hat{h}'\;} & L_G & \xleftarrow{\;\hat{r}^-\;} & L_G^- \\
{\scriptstyle m'}\downarrow & & \downarrow{\scriptstyle \hat{m}} & & \downarrow{\scriptstyle \hat{m}^-} \\
T' & \xleftarrow[\;h'\;]{} & G & \xleftarrow[\;g^-\;]{} & G^-
\end{array}
\qquad\qquad
\tag{11}
$$

12

The proof of the equivalence of the two definitions of G^- is a little more complex than its analogue for forward propagation; we give its proof here. We begin by constructing a commutative cube whose left face is the PB (8), whose front and bottom faces are respectively the PBC and PB of (9) and whose top face is the PB of (10). By the universal property of G^-, there is a unique way to complete this to a commutative cube and, by inverse pasting for PBs, the back face (and indeed the right face) is a PB:

$$(12)$$

Proposition 4.1. *The back face of (12) is a PBC.*

PROOF. Suppose we have a PB that factors through the back face:

By pasting this PB with (8), we obtain a PB that factors through the front face and, by the UP of T^-, we have a unique arrow $y_{T^-} : Y \to T^-$ such that (i) $t^- \circ y_{T^-} = h' \circ y$ and (ii) $y_{T^-} \circ f = m^- \circ \hat{h}^- \circ x^-$.

By (i), we apply the UP of G^- to obtain a unique arrow $y_{G^-} : Y \to G^-$ satisfying:

13

By diagram chase, we can apply the UP of G^- a second time to obtain a unique arrow $x_{G^-} : X \to G^-$ satisfying:

We have two candidate arrows from X to G^-: $z \circ x^-$ and $y_{G^-} \circ f$. The first satisfies (v) and (vi) immediately. The second satisfies (v) because $g^- \circ y_{G^-} \circ f = y \circ f$, by (iii), and $y \circ f = \hat{m} \circ x$; and satisfies (vi) because $h^- \circ y_{G^-} \circ f = y_{T^-} \circ f$, by (iv), and $y_{T^-} \circ f = m^- \circ \hat{h}^- \circ x^-$, by (ii).

As such y_{G^-} is the unique arrow satisfying $y = g^- \circ y_{G^-}$ and $y_{G^-} \circ f = z \circ x^-$ as required. $\qquad\square$

This establishes that the two definitions of G^- coincide. The above argument amounts to a proof of the abstract property that PBCs are stable under PBs: this requires a commutative cube whose front face is a PBC, whose left and bottom faces are PBs and where any one of the other faces is also a PB (in our case, the top face); see also Proposition 6 of [14] or Lemma 1 of [6].

4.3. The backward clean-up phase

We specify the clean-up phase by providing a mono $r^\ominus : L_G^- \leftarrow L_G^\ominus$. Clearly, and analogously to the situation for forward propagation, in order to provide such an $r^\ominus$, we already need to know L_G^-—which is dependent on the typing $G \to T'$. As such, $r^\ominus$ cannot be specified statically but should rather be provided dynamically at the time that r's rewrite is being propagated to G.

$$
\begin{array}{ccc}
L_G^- & \xleftarrow{\;r^\ominus\;} & L_G^\ominus \\
\hat{m}^- \downarrow & & \downarrow \hat{m}^\ominus \\
T^- \xleftarrow{h^-} G^- & \xleftarrow{\;g^\ominus\;} & G^\ominus
\end{array}
\qquad (13)
$$

The clean-up phase allows us to remove undesired element clones that were not specified during the strict phase of rewriting, e.g. a *partial* concept refinement where some instances of a cloned element cannot be assigned a unique type in T'. However, if $r^\ominus$ is not a mono, this phase can also create additional clones, beyond what was specified by r, and we have no use case for this extra generality, just as we have no use case for allowing the clean-up phase to add new elements to T^+ during forward propagation.

4.4. Example

We again consider an example on a multi-set. The rule $r : L \leftarrow L^-$ specifies (i) the deletion of the circle; and (ii) the cloning of the square into a (white) square and a black square. The fact that one square in G is to become white while the other becomes black is expressed by the arrow from L_G to L'.
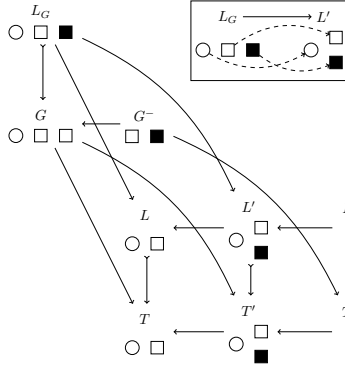
The strict phase of rewriting clones the square and retypes G, thus effecting a concept refinement; the canonical phase deletes the circle and propagates, thus deleting all circles in G.

If we have a third instance of the square in G for which we *cannot* assign a unique new type in T', we must displace the cloning operation to the second phase of rewriting and propagate to *all* instances of the square. In order to recover the same retyping of (the first two) squares as above, we must apply a clean-up rule to delete the unwanted clones.
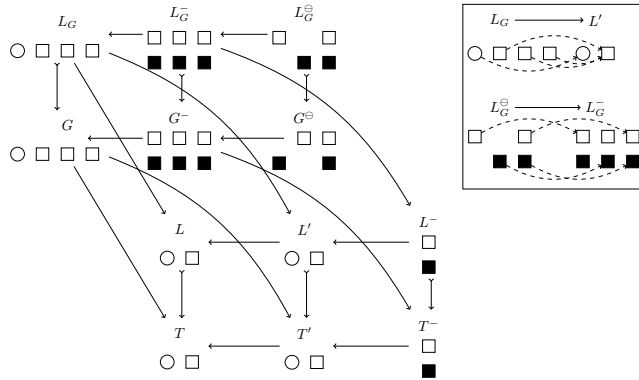
5. Rewriting general hierarchies

In this section, we consider the problem of how to update an instance of an arbitrary hierarchy and, in particular, how to do this *in-place* so that the hierarchy remains valid at all times during the propagation process.

Given a DAG H and a node s, we define the *forward sub-graph* $\vec{H}_s$ to be the largest sub-graph of H where s is the unique source node. Dually, we define the *backward sub-graph* $\overleftarrow{H}_s$ to be the largest sub-graph of H where s is the unique sink node.

In an instantiation of H in a category $\mathbf{C}$, $\vec{H}_s$ identifies all the nodes n of H whose instantiations $[\![n]\!]$ can be affected, by propagation, by an expansive rewrite of the object $[\![s]\!]$. We need not propagate an expansive rewrite of $[\![s]\!]$ to an object $[\![n]\!]$ having an arrow *into* $[\![s]\!]$ because it remains typed by post-composition with the arrow $[\![s]\!] \to [\![s]\!]^+$ induced by the rewrite.

Dually, $\overleftarrow{H}_s$ identifies all the nodes n of H that can be affected by a restrictive rewrite of $[\![s]\!]$: an object $[\![n]\!]$ having an arrow *from* $[\![s]\!]$ still types the updated object $[\![s]\!]^-$ by pre-composition with the induced arrow $[\![s]\!] \leftarrow [\![s]\!]^-$.

5.1. Expansive rewriting of a hierarchy

In section 3, we have seen how the expansive rewrite of an object G, typed by a second object T, is factorized into a strict update, producing a G' still typed by T, followed by a canonical update that produces a G^+ and propagates to T, yielding an updated T^+ that types G^+. However, in a general hierarchy, two orthogonal complications may arise: firstly, G may be typed by several graphs and their respective factorizations may be incompatible; and secondly, we may have chains of typing $G_0 \to G_1 \to \cdots G_n$ for which we need to verify the compatibility of the factorizations.

To illustrate the first point, let us consider the hierarchy $n_1 \leftarrow n_0 \to n_2$ instantiated in **Set** as follows: $[\![n_0]\!] = \emptyset$, $[\![n_1]\!] = \{\circ\}$ and $[\![n_2]\!] = \{\bullet\}$. If we update $[\![n_0]\!]$ with the rule $r : \emptyset \to \{\circ, \bullet\}$, the factorization with respect to $[\![n_1]\!]$ should specify that we add $\circ$ to $[\![n_0]\!]$ by a strict update and then add $\bullet$ and propagate this to $[\![n_1]\!]$; for $[\![n_2]\!]$, we need to do the opposite. As such, either strict update breaks one of the typing arrows. The natural solution is to propagate first, adding $\bullet$ to $[\![n_1]\!]$ and $\circ$ to $[\![n_2]\!]$, and then apply the entire rule r to $[\![n_0]\!]$ as a *strict* update—which is now possible, as an in-place update, because we have already updated $[\![n_1]\!]$ and $[\![n_2]\!]$.

For the second point, consider an instantiated hierarchy $G_0 \to G_1 \to G_2$, a rule $r : L \to L^+$ with an expansive instance $m : L \rightarrowtail G_0$ and factorizations r_1, r_1^+, x_1 and r_2, r_2^+, x_2 through L_1 and L_2 for G_1 and G_2 respectively.
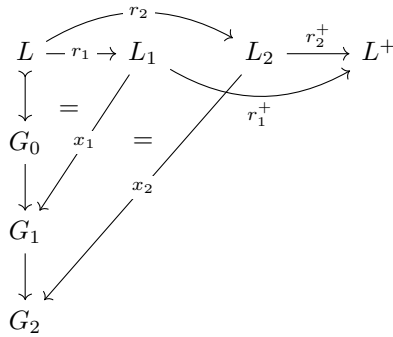
If r_1 adds an element to G_0, necessarily typed in G_1 and so in G_2 as well, but r_2 postpones this operation to r^+, therefore typing the element in G_2^+ but not G_2, it will not be possible to type G_1^+ by G_2^+. We must therefore require that r_2 extends r_1; we formalize this with the so-called *composability* condition.

Suppose we have an instantiated hierarchy $[\![H]\!]$ containing an object $G_0 = [\![n_0]\!]$ and a rule $r : L \to L^+$ with an expansive instance $m : L \rightarrowtail G_0$. For any object $G_i \in [\![\vec{H}_{n_0}]\!]$, define h_i to be the unique homomorphism from G_0 to G_i obtained by composing any path from G_0 to G_i. (They are all equal by the commutativity condition.)

Definition. Given an arrow $h_{ij} : G_i \to G_j$ in $[\![\vec{H}_{n_0}]\!]$ and factorizations

$$\tag{14}$$

that define the propagation of r to G_i and G_j respectively, we say that these factorizations are *composable* iff there exists an arrow $\ell_{ij} : L_i \to L_j$ satisfying:

$$\tag{15}$$

In the case above, if the two factorizations are compatible, we can update G_2, by taking the PO of x_2 and r_2^+, and then G_1, by taking the PO of x_1 and $r_1^+ = r_2^+ \circ \ell_{12}$. An immediate application of the UP of G_1^+, using (15), gives us the new typing $h_{12}^+ : G_1^+ \to G_2^+$.

Note that the update of G_1 is morally a *strict* update using the rule r_1^+. As written, this is not a *bona fide* rule application because x_1 is not necessarily a mono; however, this is equivalent to applying the projection of r_1 as explained in section 3. We conclude by updating G_0, by taking the PO of m and r, and obtain the retyping $h_{01}^+ : G_0^+ \to G_1^+$ as per (2).

In general, suppose we have factorizations $r_i : L \to L_i$, $r_i^+ : L_i \to L^+$ and $x_i : L_i \to G_i$ for all $G_i \in [\![\vec{H}_{n_0}]\!]$ ($i \neq 0$) and, for each $h_{ij} : G_i \to G_j \in [\![\vec{H}_{n_0}]\!]$, an arrow $\ell_{ij} : L_i \to L_j$ satisfying (15). We wish to define the updated hierarchy $[\![\vec{H}_{n_0}]\!]^+$. If $[\![\vec{H}_{n_0}]\!]$ is just G_0, we rewrite it to G_0^+ and $[\![\vec{H}_{n_0}]\!]^+$ is trivially valid. Otherwise, we define $[\![\vec{H}_{n_0}]\!]^+$ as follows:

- We update each sink node G_s to G_s^+, by taking the PO of x_s and r_s^+.

- We consider the nodes G_k that are sink nodes if we remove all the G_ss. We update each G_k, by taking the PO of x_k and r_k^+, and apply the UP of each G_k^+, using (15), to update each arrow $h_{ks} : G_k \to G_s$ to $h_{ks}^+ : G_k^+ \to G_s^+$. We then continue inductively.

This procedure updates every node and every arrow of $[\![\vec{H}_{n_0}]\!]$, i.e. it preserves the structure of the hierarchy.

Proposition 5.1. $[\![\vec{H}_{n_0}]\!]^+$ *is a valid hierarchy.*

PROOF. If $[\![\vec{H}_{n_0}]\!]^+$ is a tree, the result is immediate; so we only need to check that $[\![\vec{H}_{n_0}]\!]^+$ satisfies the commutativity condition. If there are multiple paths in $[\![\vec{H}_{n_0}]\!]^+$ from G_i^+ to G_j^+, each one comes from a distinct path from G_i to G_j in $[\![\vec{H}_{n_0}]\!]$. Each such path $G_i \to G_{i_1} \to \cdots G_{i_n} \to G_j$ gives rise to a factorization of r_i^+ as $r_j^+ \circ \ell_{i_n j} \circ \cdots \ell_{ii_1}$, by (15), and all of those paths are equal in $[\![\vec{H}_{n_0}]\!]$, by commutativity. Let us denote this path $p_{ij} : G_i \to G_j$.

By diagram chase, $\hat{r}_j^+ \circ p_{ij} \circ x_i = \hat{x}_j \circ r_i^+$ so, by the UP of G_i^+, we have a unique arrow $h_{ij}^+ : G_i^+ \to G_j^+$ satisfying $\hat{r}_j^+ \circ p_{ij} = h_{ij}^+ \circ \hat{r}_i^+$ and $\hat{x}_j = h_{ij}^+ \circ \hat{x}_i$. Each path $G_i \to G_{i_1} \to \cdots G_{i_n} \to G_j$ in $[\![\vec{H}_{n_0}]\!]$ gives rise to a path $G_i^+ \to G_{i_1}^+ \to \cdots G_{i_n}^+ \to G_j^+$ that satisfies this UP and so $[\![\vec{H}_{n_0}]\!]^+$ satisfies commutativity. $\square$

Note that the structure of $\vec{H}_{n_0}$ imposes constraints on the order in which we update the associated objects: this guarantees that, at all times during update, the entire hierarchy remains in a valid state, i.e. we can perform in-place update to obtain finally $[\![H]\!]^+$, i.e. $[\![H]\!]$ where the sub-graph $[\![\vec{H}_{n_0}]\!]$ is updated to $[\![\vec{H}_{n_0}^+]\!]$.

In the simple situation studied in section 3, clean-up can be considered as a final step of the rewrite which applies only to T and therefore cannot propagate. In a general hierarchy, a clean-up rule may itself further propagate and this requires us to specify factorizations and composability exactly as for r. As such, for the sake of avoiding redundancy, we consider it as a separate rule application and correctness follows by the above argument.

5.2. Restrictive rewriting of a hierarchy

Dually to the case of expansive rewriting, two complications may arise when we perform a restrictive update of an object G_0 in a general hierarchy: firstly, G_0 may type several objects and their respective factorizations may be incompatible; and secondly, we need to verify the compatibility of factorizations in chains of the form $G_n \to \cdots G_1 \to G_0$.

The first point typically arises in a hierarchy such as $G_1 \to G_0 \leftarrow G_2$ when we clone two nodes of G_0 and propagate one of the clones to G_1 and the other to G_2. As for the analogous case in expansive rewriting discussed above, this requires us to update G_1 and G_2 before G_0. However, in the case of restrictive rewriting, this forces us to use the lifting of the rule explicitly—because the direct construction of G_1^- and G_2^- uses G_0^-.

The second point requires us to specify composability conditions, analogous to those for expansive rewriting, in order to obtain a multi-stage factorization of r for each chain of the hierarchy.

Suppose we have an instantiated hierarchy $[\![H]\!]$ containing an object $G_0 = [\![n_0]\!]$ and a rule $r : L \leftarrow L^-$ with a restrictive instance $m : L \rightarrowtail G_0$. For any object G_i in the backward sub-graph $\overleftarrow{H}_{n_0}$, let h_i be the unique homomorphism from G_i to G_0. Given an arrow $h_{ij} : G_i \to G_j$ in $[\![\overleftarrow{H}_{n_0}]\!]$, we define L_{G_i} and L_{G_j} as in (6) and apply the UP of L_{G_j} to obtain the unique arrow $\hat{h}_{ij} : L_{G_i} \to L_{G_j}$ satisfying:

$$
\begin{array}{ccc}
 & \overset{\hat{h}_i}{\overbrace{\quad\quad}} & \\
L_{G_i} & \overset{\hat{h}_{ij}}{\dashrightarrow} L_{G_j} \overset{\hat{h}_j}{\longrightarrow} & L \\
\hat{m}_i \downarrow & = \quad\; \hat{m}_j \downarrow & \downarrow m \\
G_i & \underset{h_{ij}}{\longrightarrow} G_j \underset{h_j}{\longrightarrow} & G_0 \\
 & \underset{h_i}{\underbrace{\quad\quad}} &
\end{array}
\tag{16}
$$

The commuting triangle enables us to apply inverse pasting to establish that the commuting square is in fact a PB.

Definition. Given an arrow $h_{ij} : G_i \to G_j$ in $\overleftarrow{H}_{n_0}$ and factorizations

$$
\begin{array}{cc}
L \overset{r}{\longleftarrow} L^- & \qquad L \overset{r}{\longleftarrow} L^- \\
\hat{h}_i \uparrow \;\; r_i \;\; \downarrow r_i^- & \qquad \hat{h}_j \uparrow \;\; r_j \;\; \downarrow r_j^- \\
L_{G_i} \underset{\hat{h}_i'}{\longrightarrow} L_i & \qquad L_{G_j} \underset{\hat{h}_j'}{\longrightarrow} L_j
\end{array}
\tag{17}
$$

that define the propagation of r to G_i and G_j respectively, these factorizations are *composable* iff there exists an arrow $\ell_{ij} : L_i \to L_j$ such that

$$
\begin{array}{cc}
\begin{array}{c}
L_i \\
r_i \nearrow \;\vdots\; \nwarrow r_i^- \\
L \;= \ell_{ij} =\; L^- \\
r_j \searrow \;\downarrow\; \swarrow r_j^- \\
L_j
\end{array}
&
\qquad
\begin{array}{c}
L_i \overset{\hat{h}_i'}{\longleftarrow} L_{G_i} \\
\ell_{ij} \downarrow \quad = \quad \downarrow \hat{h}_{ij} \\
L_j \underset{\hat{h}_j'}{\longleftarrow} L_{G_j}
\end{array}
\end{array}
\tag{18}
$$

For each G_i, the lifting $\hat{r}_i : L_{G_i} \leftarrow L_{G_i}^-$ of r is defined by the PB of $\hat{h}_i'$ and r_i^-, as in (10), with instance $\hat{m}_i : L_{G_i} \rightarrowtail G_i$.

For each h_{ij}, we can connect the liftings of $\hat{r}_i$ and $\hat{r}_j$ by applying the UP of $L_{G_j}^-$, using (18), to obtain a unique arrow $\hat{h}_{ij}^- : L_{G_i}^- \to L_{G_j}^-$ satisfying:

$$
\begin{array}{ccc}
L_{G_i} & \xleftarrow{\;\hat{r}_i\;} & L_{G_i}^- \\
\scriptstyle \hat{h}_{ij} \swarrow \quad \downarrow \scriptstyle \hat{h}_i' \quad = & & \downarrow \scriptstyle \hat{h}_{ij}^- \\
L_{G_j} \xleftarrow{\;\hat{r}_j\;} & & L_{G_j}^- \; = \;\Big) \hat{h}_i^- \\
\scriptstyle \hat{h}_j' \downarrow & & \downarrow \scriptstyle \hat{h}_j^- \\
L_j \xleftarrow{\;\ell_{ij}\;} L_i \xleftarrow{\;r_i^-\;} & & L^- \\
& \xleftarrow{\;r_j^-\;} &
\end{array}
\tag{19}
$$

If we have factorizations $r_i : L \leftarrow L_i$, $r_i^- : L_i \leftarrow L^-$ and $\hat{h}_i' : L_{G_i} \to L_i$ for all $G_i \in \llbracket \bar{\breve{H}}_{n_0} \rrbracket$ ($i \neq 0$) and, for each $h_{ij} : G_i \to G_j \in \llbracket \vec{H}_{n_0} \rrbracket$, an arrow $\ell_{ij} : L_i \to L_j$ satisfying (18), we can now define the updated hierarchy $\llbracket \bar{\breve{H}}_{n_0} \rrbracket^-$. If $\llbracket \bar{\breve{H}}_{n_0} \rrbracket$ is just G_0, we rewrite it to G_0^- and $\llbracket \bar{\breve{H}}_{n_0} \rrbracket^-$ is trivially valid. Otherwise, we define it as follows:

- We update each source node G_s to G_s^-, by taking the PBC of $\hat{r}_s$ and $\hat{m}_i$.

- We consider the nodes G_k that are source nodes if we remove all the G_ss. We update each G_k, by taking the PBC of $\hat{r}_k$ and $\hat{m}_k$, and apply the UP of each G_k^+, using (18), (16) and (19) as in (11), to update each arrow $h_{sk} : G_s \to G_k$ to $h_{sk}^- : G_s^- \to G_k^-$. We then continue inductively.

This procedure updates every node and every arrow of $\llbracket \bar{\breve{H}}_{n_0} \rrbracket$, i.e. it preserves the structure of the hierarchy. The following diagram shows the general situation for an arrow h_{ij}.

$$
\tag{20}
$$

Proposition 5.2. $[\![\vec{H}_{n_0}]\!]^-$ *is a valid hierarchy.*

PROOF. We only need to check the commutativity condition. Each path in $[\![\bar{H}_{n_0}]\!]^+$ from G_i^- to G_j^- comes from a distinct path from G_i to G_j in $[\![\bar{H}_{n_0}]\!]$. Each such path $G_i \to G_{i_1} \to \cdots G_{i_n} \to G_j$ gives rise to a factorization of r_j^- as $\ell_{i_n j} \circ \cdots \ell_{i i_1} \circ r_i^-$, by (18), and all of those paths are equal in $[\![\vec{H}_{n_0}]\!]$, by commutativity. Let us denote this path $p_{ij} : G_i \to G_j$.

By the UP of G_j^-, as in (20), we have a unique arrow $h_{ij}^- : G_i^- \to G_j^-$ satisfying $p_{ij} \circ g_i^- = g_j^- \circ h_{ij}^-$ and $h_{ij}^- \circ \hat{m}_i^- = \hat{m}_j^- \circ \hat{h}_{ij}^-$. Each path $G_i \to G_{i_1} \to \cdots G_{i_n} \to G_j$ in $[\![\bar{H}_{n_0}]\!]$ gives rise to a path $G_i^- \to G_{i_1}^- \to \cdots G_{i_n}^- \to G_j^-$ that satisfies this UP and so $[\![\bar{H}_{n_0}]\!]^-$ satisfies commutativity. $\qquad\square$

As for expansive rewriting of a hierarchy, the structure of $\bar{H}_{n_0}$ constrains the order in which we update to guarantee that, at all times during update, the entire hierarchy remains in a valid state, i.e. we can perform in-place update to obtain finally $[\![H]\!]^-$, i.e. $[\![H]\!]$ where the sub-graph $[\![\bar{H}_{n_0}]\!]$ is updated to $[\![\bar{H}_{n_0}^+]\!]$.

6. Implementation and use cases

6.1. The `ReGraph` Python library

In the preceding sections, we have detailed the mathematical theory of SqPO rewriting in general hierarchies. We have implemented this theory—at the time of writing, for the setting of *simple* directed graphs with attributes on nodes and edges, although there is no conceptual or technological problem to extend this to directed *multi*-graphs—in the `ReGraph` Python library[1]. The library provides two back-ends: in-memory graphs, based on the `networkX` library widely used in complex systems—and persistent graphs using the Neo4j graph DB. Rules can be expressed declaratively, essentially using the mathematical definition used in this paper, or procedurally, using a simple language to express the primitive operations of clone, delete, add and merge.

Hierarchies, rewriting and propagation are implemented natively, in Python, for the in-memory back-end. However, the implementation of the persistent back-end requires a considerably greater effort because (i) Neo4j currently only provides a *single* graph within which we must encode an arbitrary hierarchy; and (ii) all operations to be performed on that graph must be expressed through the Cypher query language used by Neo4j. The precise details of the encoding and the translation of rules into Cypher are highly technical and not of great interest in their own right. However, their ultimate effect is to enforce an *abstraction barrier* that gives the illusion that Neo4j actually provides an implementation of our theory and, provided that the user only accesses Neo4j through `ReGraph`, guarantees that the current contents of the single underlying Neo4j graph always corresponds to a valid encoding of a hierarchy.

---

[1] https://github.com/Kappa-Dev/ReGraph

The principal difference between the theory presented in this paper and the implementation lies in the specification of propagation: in `ReGraph`, a controlled propagation is specified by a single relation that plays the same rôle as the strict and clean-up phases presented here. For example, the partial concept refinement of section 4.5 is expressed by the same rule together with a relation that specifies, for the first two squares, how to retype them; nothing needs to be specified for the third square which, as a result, is cloned.

6.2. Multi-set rewriting

The simplest non-trivial hierarchy consists of two objects connected by a single arrow. Note that, in this case, the hierachy coincides with its skeleton. The instantiation of this hierarchy in **Set** specifies a set T of *types* and a set G of *instances* of these types where the typing of the elements of T is given by the function from G to T; as such, this provides an intensional representation of a multi-set over the set T.

The usual notion of multi-set rewriting operates only on G—the set T is fixed in advance—but our framework provides a rigorous framework within which rewrites can also apply to T—either directly or, in practice more likely, through the forward propagation of a rewrite of G. In this way, the set of types can grow automatically, on the fly, an approach that otherwise requires a substantial algorithmic and implementation effort. However, for our implementation to provide an efficient simulation engine for such systems, further development would be necessary in order to exploit the intrinsic *causality* between rules—which possible rule applications are created and destroyed by the application of a given rule—in order to maintain and update incrementally and efficiently the collection of all possible rule applications at any given time.

6.3. Graph databases

Most modern graph DBs, such as Neo4j, are based on the notion of property graphs, i.e. directed (multi-)graphs where nodes and edges may have a dictionary of *properties* consisting of *key-value* pairs. If we instantiate the hierarchy $e : d \to s$ as $h : G \to T$ in the category of property graphs, we obtain a setting where the graph T defines the permitted node and edge types as well as all possible properties, i.e. T acts as a basic kind of *schema*. The graph G defines the *data* graph and the homomorphism to T specifies the types of all its nodes and edges and guarantees that all edges and properties *validate* the schema.

The implementation of `ReGraph` with persistent back-end thus provides the illusion of a notion of schema for Neo4j graphs. Let us underline our dual use of Neo4j: on the one hand, we exploit it simply as one possible back-end technology for `ReGraph`; on the other, we exploit our theory to provide it with a notion of sachema. In fact, we have also made an optimized implementation for this particular hierarchy which therefore avoids most of the overheads associated with the encoding into the single underlying Neo4j graph[2].

---

[2]https://github.com/Kappa-Dev/ReGraph/blob/master/regraph/neo4j/graphs.py

In order to build a fully general front-end to Neo4j in this way, we need to extend `ReGraph` to work with non-simple graphs; we plan to do this in the near future. However, our work can also been viewed as a proposal for how to incorporate schema, or indeed multiple graphs, *natively* within a graph DB such as Neo4j rather than merely a means of encoding this. Indeed, the recently-launched ISO standardization process for GQL, and the associated informal working group PGSWG, is investigating ways to express multiple graphs and/or schema graphs. The extent to which our ideas are ultimately reflected in GQL will determine the extent to which our current encoding and implementation can be simplified and rendered more efficient.

In this setting, forward propagation automatically constructs and applies an update of the schema graph in the light of an update of the data graph that would otherwise have broken schema validation, i.e. a descriptive update. Such updates typically occur during the earlier phases of application development where we do not yet have a clear picture of the structure of all the relevant data and therefore wish to allow the schema to evolve dynamically to accommodate the incoming data. Dually, backward propagation constructs an automatic update of the data graph in the event of an update of the schema, i.e. what is known as a prescriptive update. Such updates more usually occur later in the development process where we wish to engineer specific refinements to the schema in the light of the needs of the application. This point is discussed in more detail in [2].

Let us give here an example of a *concept refinement* where a node of the schema is cloned into finer-grained concepts by the application of a restrictive rule $L \leftarrow L^-$ as in figure 1. The rule matches the node *Message* in the original schema S and clones it into two nodes, *Post* and *Comment*, producing a new schema graph S^- with the concept of 'message' replaced by the finer-grained concepts of 'post' and 'comment'.
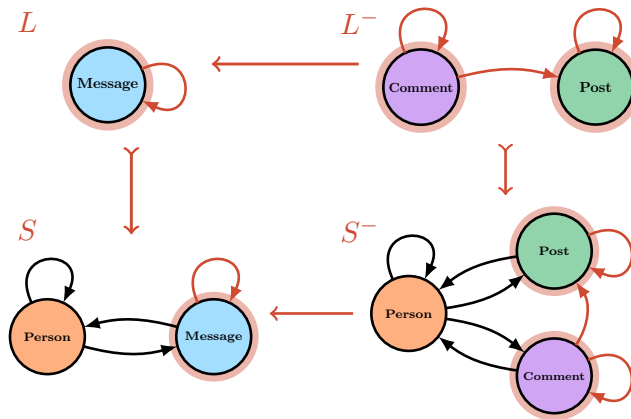

Figure 1: Example of a restrictive rewrite of a schema graph.

A data graph validated by the original schema S is no longer validated by S^- because we no longer have the means to type *Message* nodes. Canonical backward propagation would duplicate each *Message* node into a *Post* node and a *Comment* node; however, in a concept refinement, we seek rather to reattribute a unique type to each node of the data graph. This is specified by a backward factorization (17) as depicted in figure 2. The homomorphism $f : G \to S$ is encoded by node colours; nodes highlighted with orange represent the sub-object L_G whose typing is affected by the rewriting of the schema—in this case, all the instances of *Message*. The orange dashed arrows in the figure represent the mapping of nodes defining the arrow $L_G \to L^-$ so that n_6 and n_2 are to become comments and n_5 a post.
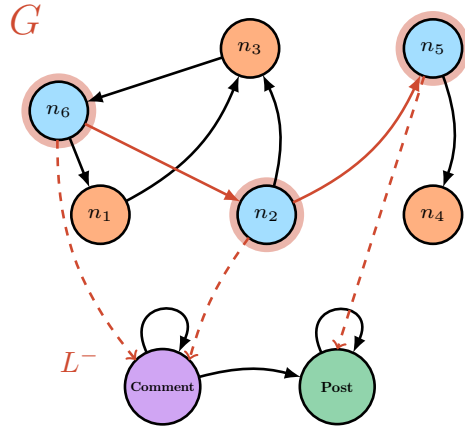

Figure 2: Typing of the affected data subgraph by the rewritten schema.

The arrow $L_G \to L^-$ specifies a purely strict update of G, i.e. no nodes of G are cloned, only their types are modified. The resulting retyped version of G is depicted in figure 3 where the new node colours encode the typing by S^-.
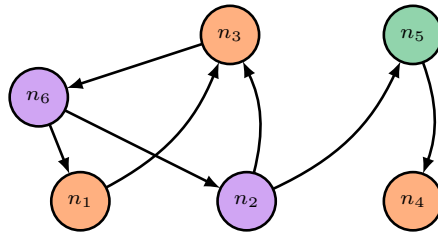

Figure 3: Final retyping of the data graph by the refined schema.

This example is based on the LDBC Social Network Benchmark[3]. In order to illustrate the role of the composability condition, let us augment the example with a richer hierarchy containing an additional node x and edges $d \to x$ and $x \to s$ and instantiate this as in figure 4. The new graph $\hat{S}$, the *semantic schema*, represents certain kinds of knowledge about our data, i.e. that a 'person' is a Facebook or Twitter profile, a 'message' is either a Facebook post, Facebook comment, tweet or a reply to a tweet. The data graph is typed by the semantic schema, as well as the original schema, and we encode this typing with different node shapes in the figure. By definition, the triangle commutes.
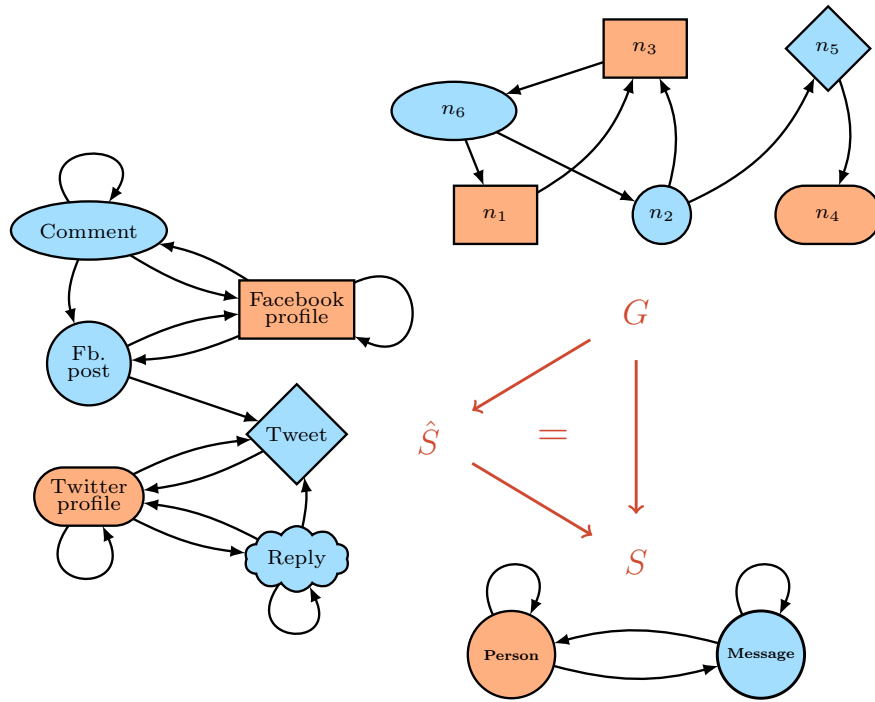

Figure 4: A 'triangle' hierarchy representing the social network data, schema and semantic schema.

Recall the restrictive rule applied in the previous example that refines the concept of a 'message' into the finer-grained concepts of 'post' and 'comment'. If we factorize this rule for G as above and for $\hat{S}$ by assigning *Comment* and *Reply* to *Comment* (in S^-) and *Fb. post* and *Tweet* to *Post*, we obtain an invalid instance of the hierarchy as in figure 5: the data node n_2 in the figure becomes typed by *Comment* in the schema, while being typed as *Fb. post* in the semantic schema, which in turn is typed as *Post* by the schema.

---

[3]https://ldbc.github.io/ldbc_snb_docs/ldbc-snb-specification.pdf

This implies that the two backward propagations are not composable. To see why, consider the nodes and edges of $\hat{S}$ and G highlighted with orange in figure 5. The highlighted elements represent the sub-objects $L_{\hat{S}}$ and L_G of $\hat{S}$ and G respectively whose typing is affected by the rewriting of S. It is the easy to verify that the given arrows do not respect the composability condition (18).
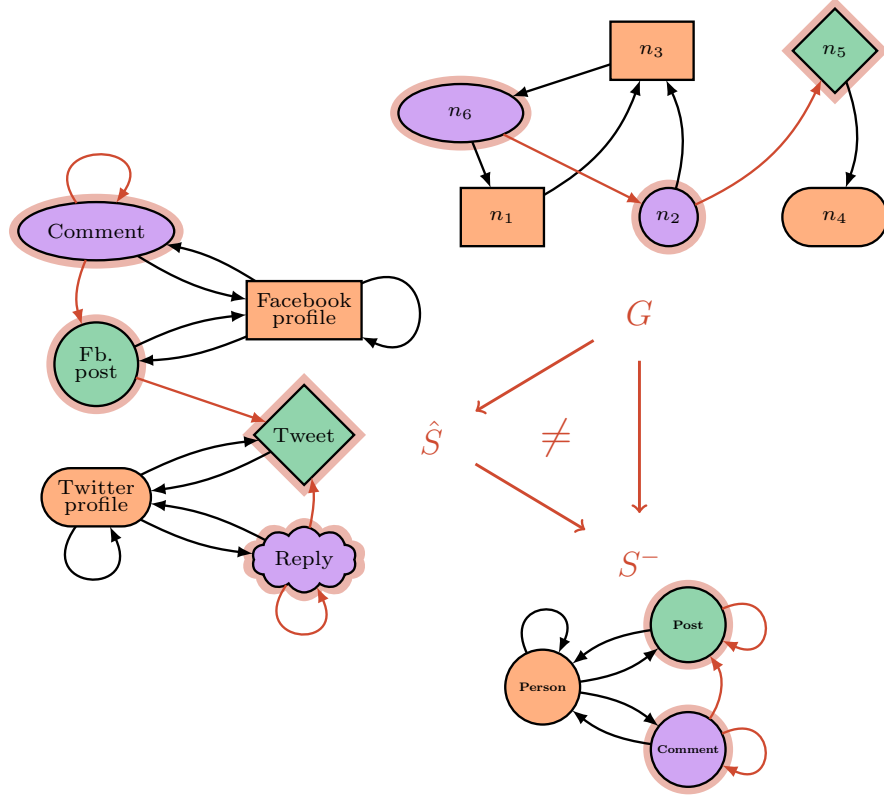

Figure 5: The resulting invalid hierarchy instance.

6.4. The KAMI bio-curation system

The original motivating use case for the development of the theory presented in this paper was the KAMI bio-curation system which provides a graph-based KR for protein-protein interactions (PPIs) in cellular signalling. The skeleton of KAMI's hierarchy consists of a chain of three nodes and two edges: $N \to A \to M$ where M represents the *meta-model*, a graph defining the basic concepts of the system such as proteins, binding sites and binding interactions; A represents the *action graph*, which defines the particular proteins and interactions in a knowledge corpus; and N represents the *nuggets*, small graphs that capture the necessary conditions for a PPI to occur. A typical KAMI hierarchy has multiple nuggets but a single action graph and meta-model.

Unlike the previous use case, for graph DBs, the hierarchy itself can evolve over time: this usually occurs upon the addition of a completely new nugget but can also arise if a nugget is deleted or two nuggets are merged together (or even if a nugget is cloned although we have not yet needed to consider this case in practice). Such updates operate on the *structure* of the KR rather than on its *contents* and therefore lie out of the scope of this paper.

KAMI[4] [13, 11] is built on top of the ReGraph library. It makes extensive use of forward propagation in order to aggregate new PPIs appropriately into an existing knowledge corpus: if it identifies that a node mentioned in an input nugget already exists in the action graph, it constructs a strict rewrite, to reuse that node, rather than creating a new one by canonical propagation.

For example, a new nugget is first added to the hierarchy as an empty graph N to which an expansive rule, such as $L \to L^+$ in figure 6, is applied through the trivial instance. In order to propagate this update to the action graph, KAMI identifies the 'pieces of knowledge' *already* present in the action graph, i.e. the entities and actions that have already been encountered by the system, to construct the object L' defining a factorization of the original rule and its typing by the action graph. In the figure below, L' is depicted as the sub-object of L^+ highlighted with orange and its typing by the action graph is depicted with the orange dashed arrows.
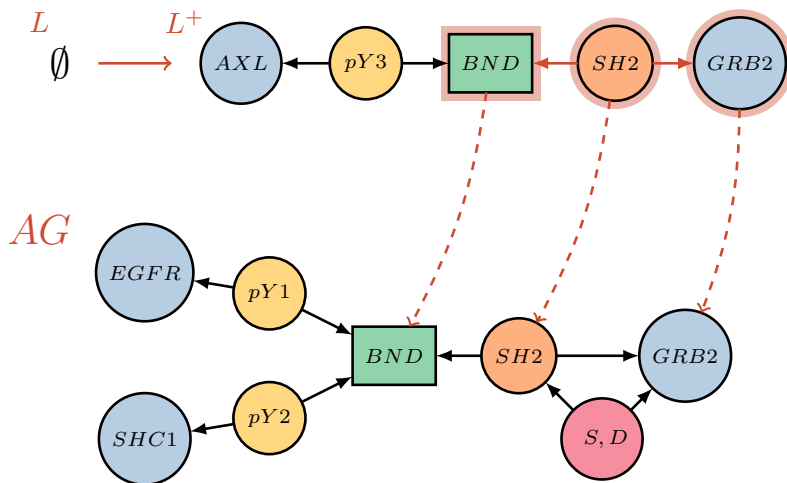

Figure 6: Partial typing of the right-hand side of the action graph by the schema.

As a result of rewriting and propagation, the new 'pieces of knowledge' are added to the action graph, corresponding to the highlighted elements of the updated action graph AG^+ in figure 7, so that the new nugget N^+ is typed by this updated action graph.

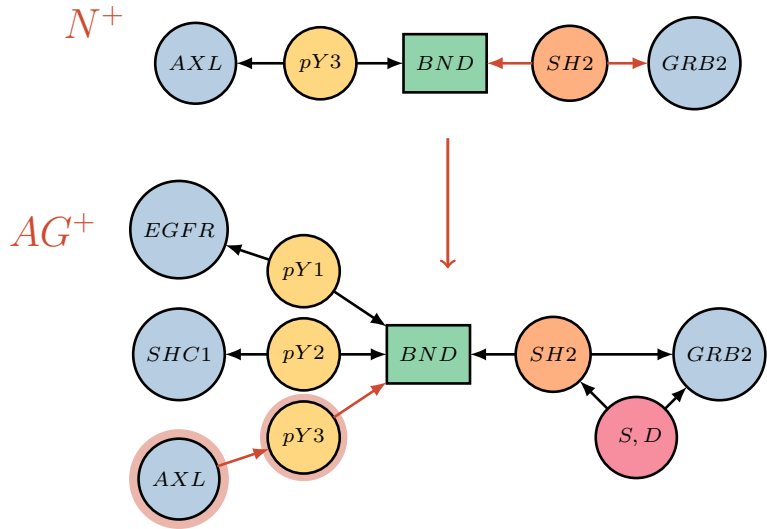
[4]https://github.com/Kappa-Dev/KAMI

Figure 7: Result of knowledge aggregation.

On the other hand, the meta-model of KAMI is required to remain invariant under all update operations. In other words, any update that propagates to the meta-model must be strict. In practice, this means that all new elements of a nugget must have a well-defined type in the meta-model—even if they do not exist in the action graph—so that, in this example, L^+ must be entirely typed by the meta-model in a way that is compatible with the typing of elements of L'. This requirement is an instance of the composability condition.

KAMI also makes use of backward propagation in order to contextualize knowledge to a particular collection of gene products: a restrictive update of the action graph propagates to the nuggets from which we can determine automatically which nuggets actually apply to a particular gene product [11].

7. Conclusions

We have presented a formalism for graph-based knowledge representation and update that exploits SqPO rewriting to perform updates anywhere in a hierarchy of objects (typically sets or graphs). As in the extended abstract of this paper [12], we have chosen a rigorous, but largely informal, presentation. The main contribution can be stated as follows: given a hierarchy, a SqPO rule, an expansive (resp. restrictive) instance of that rule into some object O and factorizations for *every* other object on a path from (resp. to) O satisfying composability, we can *uniquely* rewrite the entire hierarchy in a way that guarantees the validity of the result, i.e. so that the updated hierarchy still respects the commutativity condition.

The requirement to specify all these factorizations—and also verify that they satisfy composability if necessary—can, in principle, be very onerous. However, although our definitions allow for arbitrarily complicated hierarchies, in practice we have yet to encounter the need for particularly deep hierarchies (three or four levels seems enough for all use cases we have considered). Furthermore, our experience suggests that most updates need propagate only along single edges or, at most, paths of length two so that, in practice, the requirement does not seem too onerous.

The other principal open question concerns the characterization of the data structures necessary to maintain an *audit trail* of all updates made to a system. This would enable us to determine whether an update can be undone or not, a question that is greatly complicated by the fact of propagation, and, more generally, provide support for maintaining different *versions* of the contents of a KR. This requires a major generalization of the theory of *causality* between SqPO rules; see [10] for example. We intend to investigate this question first in the two concrete use cases discussed in this paper before attempting a full-blown generalization to arbitrary hierarchies.

References

[1] Baader, F., Calvanese, D., McGuinness, D., Patel-Schneider, P., Nardi, D.: The Description Logic handbook: theory, implementation and applications. CUP (2003)

[2] Bonifati, A., Furniss, P., Green, A., Harmer, R., Oshurko, E., Voigt, H.: Schema validation and evolution for graph databases. In: Proceedings of the 38th International Conference on Conceptual Modeling (ER) (2019)

[3] Chen, P.P.S.: The entity-relationship model—toward a unified view of data. ACM Transactions on Database Systems (TODS) **1**(1), 9–36 (1976)

[4] Corradini, A., Heindel, T., Hermann, F., König, B.: Sesqui-pushout rewriting. In: Proceedings of the 3rd International Conference on Graph Transformation (ICGT). pp. 30–45. Springer (2006)

[5] Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M.: Algebraic approaches to graph transformation–part I: Basic concepts and double pushout approach. In: Handbook Of Graph Grammars And Computing By Graph Transformation: Volume 1: Foundations, pp. 163–245. World Scientific (1997)

[6] Danos, V., Heindel, T., Honorato-Zimmer, R., Stucki, S.: Reversible sesqui-pushout rewriting. In: Proceedings of the 7th International Conference on Graph Transformation (ICGT). pp. 161–176. Springer (2014)

[7] Dyckhoff, R., Tholen, W.: Exponentiable morphisms, partial products and pullback complements. Journal of Pure and Applied Algebra **49**(1-2), 103–116 (1987)

[8] Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A., Corradini, A.: Algebraic approaches to graph transformation–part II: Single pushout approach and comparison with double pushout approach. In: Handbook Of Graph Grammars And Computing By Graph Transformation: Volume 1: Foundations, pp. 247–312. World Scientific (1997)

[9] Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., Plantikow, S., Rydberg, M., Selmer, P., Taylor, A.: Cypher: An evolving query language for property graphs. In: Proceedings of the 2018 International Conference on Management of Data (SIGMOD). pp. 1433–1445. ACM (2018)

[10] Harmer, R.: Rule-based meta-modelling for bio-curation. Habilitation à Diriger des Recherches, ENS Lyon, France (2017)

[11] Harmer, R., Oshurko, E.: KAMIStudio: An environment for biocuration of cellular signalling knowledge. In: International Conference on Computational Methods in Systems Biology (CMSB). pp. 322–328. Springer (2019)

[12] Harmer, R., Oshurko, E.: Knowledge representation and update in hierarchies of graphs. In: Proceedings of the 12th International Conference on Graph Transformation (ICGT). pp. 141–158. Springer (2019)

[13] Harmer, R., Le Cornec, Y.S., Légaré, S., Oshurko, E.: Bio-curation for cellular signalling: the KAMI project. IEEE/ACM transactions on computational biology and bioinformatics **16**, 1562–1573 (2019)

[14] Löwe, M.: Graph rewriting in span-categories. In: Proceedings of the 5th International Conference on Graph Transformation (ICGT). pp. 218–233. Springer (2010)