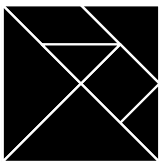
GSDLAB TECHNICAL REPORT

Intermodeling, queries and Kleisli categories

Zinovy Diskin

GSDLAB–TR 2011–10–01 October 2011



Generative Software
Development Lab

Generative Software Development Laboratory
University of Waterloo
200 University Avenue West, Waterloo, Ontario, Canada N2L 3G1

WWW page: http://gsd.uwaterloo.ca/

Intermodeling, queries and Kleisli categories*

Zinovy Diskin[1,2]

[1] NECSIS, McMaster University, Canada
[2] Generative Software Development Lab,
University of Waterloo, Canada

diskinz@mcmaster.ca

Abstract. Specification and maintenance of relationships between models are vital for MDE. We show that a wide class of such relationships can be specified in a compact and precise manner if intermodel mappings involve derived model elements computed by corresponding queries. Composition of such mappings is not straightforward and requires specialized algebraic machinery. We present a formal framework, in which such machinery can be generically defined for a wide class of metamodel definitions, and thus important intermodeling scenarios can be algebraically specified.

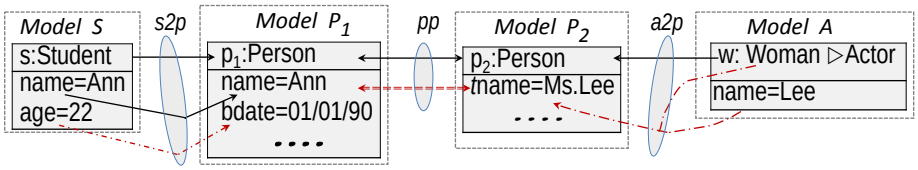
1 Introduction

Model-driven engineering (MDE) is a prominent approach to software development, in which models of the domain and the software system are primary assets of the development process. Normally models are inter-related, perhaps in a very complex way, and to keep them consistent and use them coherently, relationships between models must be accurately specified and maintained. As noted in [1], "development of well-founded techniques and tools for the creation and maintenance of intermodel relations is at the core of MDE." The goal of our paper is to present a theoretical framework for specifying a wide class of intermodel relationships in a compact and formal way. Then many practically useful operations over inter-related models can be algebraically specified.

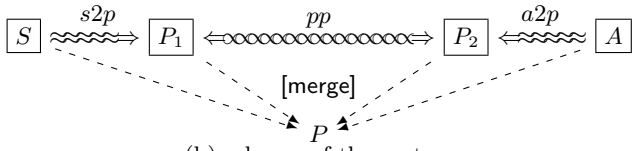
To illustrate the issues we are going to address, let us consider a simple example of model integration in Fig. 1. Subfigure (a) presents four object models. The expression o:Name in the upper compartment of a model box declares an objects o of class Name; the lower compartment shows o's attribute values, and ellipses in models P_1, P_2 refer to other attributes not shown. In model A, class Woman extends class Actor. When we need to refer to an element e (an object or an attribute) of model X, we write $e@X$. Arrows between models denote intermodel relationships explained below.

We suppose that models P_1 and P_2 are developed by two different teams charged with specifying different aspects of the same domain—different attributes of the same person in our case. Bidirectional arrow between objects $p_1@P_1$ and

* Minor revision in November 2012



(a) four models linked informally



(b) schema of the system

Fig. 1. Running example: four models and their relationships, informally

$p_2@P_2$ means that these objects are different representations of the same person. Model P_1 gives the first name whereas P_2 provides the last name and the title of the person ('tname'). We thus have a complex relationship between the attributes shown by a dashed link (brown with a color display): both attributes talk about names but are complementary. Together, the two links form an informal mapping *pp* between the models.

We also assume that model P_1 is supplied with a secondary model S representing a specific view of P_1 to be used and maintained locally at its own site (in the database jargon, S is a materialized view of P_1). Mapping $s2p$ consisting of three links defines the view informally. Two solid-line links declare "the sameness" of the respective elements. The dash-dotted link shows relatedness of the two attributes but says nothing more. Similarly, mapping $a2p$ is assumed to define model A as a view to model P_2: the solid link declares the "sameness" of the two objects, and the dash-dotted link shows relatedness of their attributes and types. Mappings $s2p$, *pp* and $a2p$ bind all models together so that a virtual integrated (or merged) model, say P, should say that Ms. Ann Lee is a student and actor born on Jan 1, 1990. Diagram Fig. 1(b) presents the merge informally: horizontal fancy arrows denote intermodel mappings, and dashed inclined arrows show mappings that embed the models into the merge.

Building model management tools capable to perform integration like above for industrial models (normally containing thousands of elements) requires clear and precise specifications of intermodel relationships. Hence, we need a framework in which intermodel mappings could be specified formally, then operations on models and model mappings could be described in precise algebraic terms. For example, merging would appear as an instance of a formal operation that takes a diagram of models and mappings and produces an integrated model together with embeddings as shown in Fig. 1(b). We want such descriptions to be generic and applicable to a wide class of scenarios over different metamodels. Category theory does provide a suitable methodological framework (cf. [2–4]), e.g., homogeneous merge can be defined as the colimit of the corresponding diagram [5, 6],

and heterogeneity can be treated as shown in [7]. However, the basic prerequisite for applying categorical methods is that mappings and their composition must be precisely defined. It is not straightforward even in our simple example, and we will briefly review the problems to be resolved.

Thinking in terms of elements, a mapping should be a set of links between models' elements as shown by ovals in Fig. 1(a). We can consider a link formally as a pair of elements, and it works for those links in Fig. 1(a), which are shown with solid lines; semantically, such a link means that two elements represent the same entity in the real world. However, we cannot declare attributes 'age' in model S (we write 'age'@S) and 'bdate'@P_1 to be "the same" because they are related yet different. Even more complex is the relationship between attribute 'tname' (name with title) in base model P_2 and the view model A: it involves attributes and types (the Woman-Actor subclassing) and is shown informally by a two-to-one dash-dotted link. Finally, the dashed link between elements 'name'@P_1 and 'tname'@P_2 encodes a great deal of semantic information described above.

In the literature, indirect relationships like above are usually specified by *correspondence rules* [8] or *expressions* [9] attached to the respective links. When such annotated links are composed, it is not clear how to compose the rules; hence, it is difficult to manage scenarios that involve composition of intermodel mappings. The importance and difficulty of the mapping composition problem is well recognized in the database literature [9], we think it will also become increasingly important in software engineering with advancing and maturing MDE methods. The main goal of the paper is to demonstrate that the problem can be solved by standard means of categorical algebra (which are, however, to be applied in a non-standard way).

In more detail, the paper makes three contributions. First, we show that for a wide class of intermodel relationships, informal mappings as in Fig. 1(b) can be presented by combinations of simple formal mappings (functions): each one goes from a source model to a target model and consists of pairs of models' elements. The key idea is that we allow target elements in such pairs to be *derived* rather than *basic* elements of the target model, that is, be results of some operations performed with basic elements; the next section will explain this in detail. We call such operations *queries*; the reader may think of some predefined query language that determines a class of legal operations and the respective derived elements. We will call links and mappings involving queries *q-links* and *q-mappings*. Q-links allows us to eliminate multi-ary links and replace them by binary links targeting at elements derived with multi-ary queries.

Although for this paper model merge is a sample intermodeling scenario rather than a central point, we believe that clarity and compactness of the approach based on q-links and q-mappings is itself an important contribution. It proves to be instrumental for specifying and guiding the difficult problem of model merge.

Third, we build a mathematical framework, in which q-mappings can be formally specified and modeled algebraically (the *q-framework*). An important

feature of the framework is that models and q-mappings are structured as a category, and so operations with models and mappings can be specified by standard algebraic means developed in category theory. In more detail, we model a query language by a *monad* (a well-known constructs of categorical algebra) , and then q-mappings can be formalized as *Kleisli morphisms* of the monad (see [10] for a brief exposition). Thus, intermodeling scenarios can be placed into respective Kleisli categories and become amenable to algebraic treatment.

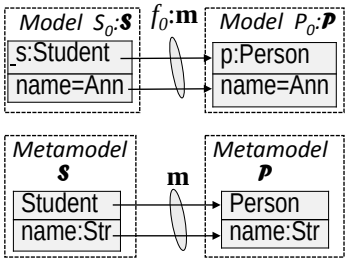
The structure of the paper is as follows. In Section 2 we consider our running example in more detail, and show how the idea of q-links and q-mappings works. We also demonstrate important issues to be addressed in the q-framework. Section 3 explains the main points of the formalization: models' conformance to metamodels, retyping, and the query mechanism and q-mappings. Section 4 describes related work and Section 5 concludes. Due to space limitations, several important discussions and mathematical explanations are omitted, but can be found in our technical report [11] accompanying the paper. Below we will refer to it as "the TR". Particularly, definitions of categorical notions not defined in the paper can be found in the TR.

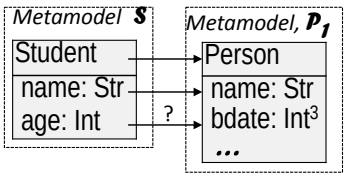
2 Intermodeling and Kleisli mappings

We consider our running example and consecutively introduce main features of our specification framework.

2.1 From informal to formal mappings

Type discipline. Before mapping models we need to map their metamodels. Suppose we need to match models S_0 and P_0 over corresponding metamodels $\boldsymbol{\mathcal{S}}$ and $\boldsymbol{\mathcal{P}}$ resp. (see the inset figure on the right), and link objects $s@S_0$ and $p@P_0$ as being "the same". However, these objects have different types ('Student' and 'Person' resp.) and, with a strict type discipline, cannot be matched. Indeed, the two objects can only be "equaled" if we know that their types actually refer to the same, or, at least, not disjoint, classes of real world objects. For simplicity, we assume that classes Student@$\boldsymbol{\mathcal{S}}$ and Person@$\boldsymbol{\mathcal{P}}$ refer to the same class of real world entities but are named differently; and their attributes 'name' also mean the same. To make this knowledge explicit in our specification, we match the metamodels $\boldsymbol{\mathcal{S}}$ and $\boldsymbol{\mathcal{P}}$ via mapping $\boldsymbol{m}$ as shown in the inset figure. After metamodels are matched, we can type-safely match objects s and p, and their attributes as well. The notation f_0:$\boldsymbol{m}$ means that each link in mapping f_0 is typed by a corresponding link in mapping $\boldsymbol{m}$. Below we will often omit metamodel postfixes next to models and model mappings if they are clear from the context.



Indirect linking, queries and q-mappings. 

As argued above, to specify relationships between models S and P_1 in Fig. 1, we first need to relate their metamodels (the inset figure on the right). However, we cannot directly relate attributes 'age' and 'bdate'. The cornerstone of our approach to intermodeling is the idea to specifying indirect relationships by *direct* links to derived elements computed with suitable queries. For example, attribute 'age' can be derived from 'bdate' with an evident query $Q1$:

$$/\text{age} = Q1(\text{bdate}) = 2011 - \text{bdate.byear},$$

where we follow UML and prefix the names of derived elements by slash, $Q1$ is the name of the query, and the last term is its body (definition); 'byear' denotes the year-field of the bdate-records. Now the relation between metamodels $\boldsymbol{\mathcal{S}}$ and $\boldsymbol{\mathcal{P}}_1$ is specified by three directed links, i.e., pairs, (Student, Person), (name, name) and (age, /age) as shown in Fig. 2(a) (basic elements are shaded whereas the derived attribute '/age' is blank). The three links form a *direct* mapping $\boldsymbol{m_1}\colon \boldsymbol{\mathcal{S}} \to \boldsymbol{\mathcal{P}}_1^+$, where metamodel $\boldsymbol{\mathcal{P}}_1^+$ denotes $\boldsymbol{\mathcal{P}}_1$ augmented with derived attribute /age. Since mapping $\boldsymbol{m_1}$ is total, it indeed defines metamodel $\boldsymbol{\mathcal{S}}$ as a view of $\boldsymbol{\mathcal{P}}_1$.

Query $Q1$ can be executed for any model over metamodel $\boldsymbol{\mathcal{P}}_1$, in particular, P_1 (Fig. 2(a) top), which results in augmenting model P_1 with the corresponding derived element; we denote the augmented model by P_1^+. Now model S can be directly mapped to model P_1^+ as shown in Fig. 2(a), and each link in mapping f_1 is typed by a corresponding link in mapping $\boldsymbol{m_1}$.

The same idea works for specifying mapping $a2p$ in Fig. 1. The only difference is that now derived elements are computed by a more complex query (with two *select-from-where* clauses) as shown in Fig. 2(b): mapping $\boldsymbol{m_2}$ provides a view definition, which is executed for model P_2 and results in view model A and traceability mapping f_2. In this way we formalize informal mappings $s2p$, $a2p$ in Fig. 1 by formal mappings into models and metamodels augmented with derived elements. Recall that we will call such mappings *q-mappings*. Note that ordinary mappings can be seen as degenerate q-mappings that do not use derived elements.

Links-with-new-data via spans. Relationships between models P_1 and P_2 in Fig. 1 were informally explained in Introduction. A more precise description is given by Fig. 3. We first introduce a new metamodel $\boldsymbol{\mathcal{P}}_{12}$ (the shaded part of metamodel $\boldsymbol{\mathcal{P}}_{12}^+$), which specifies new concepts assumed by the semantics. Then we relate these new concepts to the original ones via mappings $\boldsymbol{r}_1$, $\boldsymbol{r}_2$; the latter one essentially uses derived elements. Queries $Q4_{1,2}$ are projection operations, and query Q_3 is the pairing operation. Particularly, mapping $\boldsymbol{r}_2$ says that attribute 'fname'@$\boldsymbol{\mathcal{P}}_{12}^+$ does not match any attribute in model $\boldsymbol{\mathcal{P}}_2^+$, 'lname'@$\boldsymbol{\mathcal{P}}_{12}^+$ is the same as '/name'@$\boldsymbol{\mathcal{P}}_2^+$ (i.e., the second component of 'tname'), and 'tname'@$\boldsymbol{\mathcal{P}}_2^+$ "equals" to the pair of attributes (title, lname) in $\boldsymbol{\mathcal{P}}_{12}^+$.

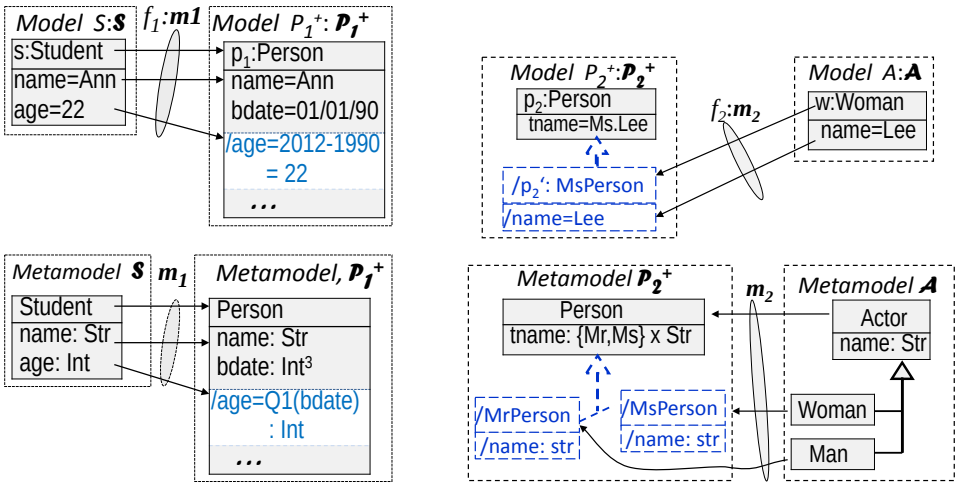


Fig. 2. Indirect matching via queries and direct mappings

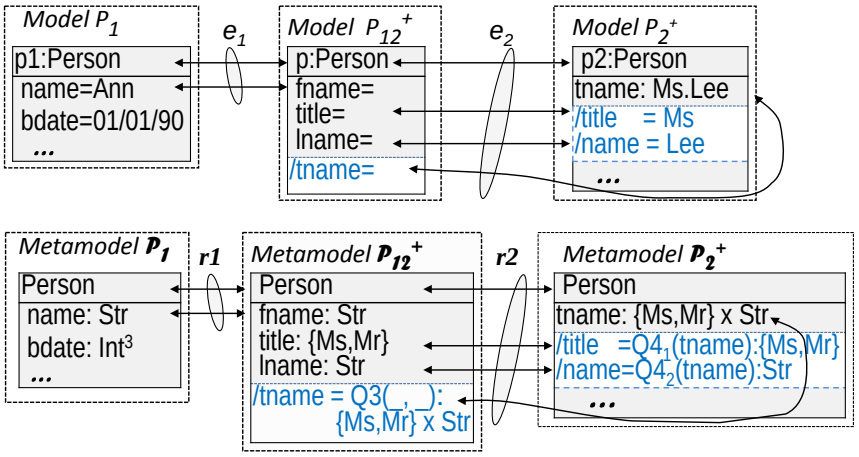


Fig. 3. Matching via spans and queries

On the level of models, we introduce a new model P_{12} to declare sameness of objects $p_1@P_1$ and $p_2@P_2$, and to relate their attribute slots. The new attribute slots are kept empty—they will be filled-in with the corresponding local values during the merge.

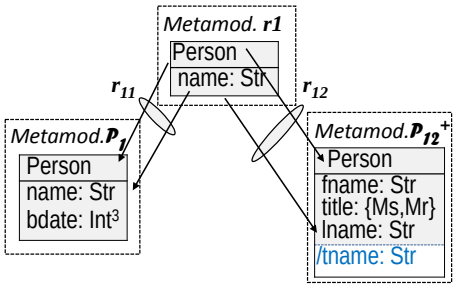
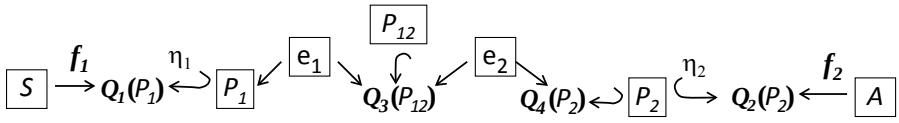


Fig. 4. Partial mappings via spans

It is well-known that the algebra of totally defined functions is much simpler than of partially defined ones. Neither of the mappings $\boldsymbol{r}_k$, e_k $(k = 1, 2)$ is total (recall that $\boldsymbol{\mathcal{P}}_2$ and P_2 may contain other attributes not shown in our diagrams). To replace these partial mappings with total ones, we apply a standard categorical construction called a *span*, as shown in Fig. 4 for mapping $\boldsymbol{r}_1$. We reify $\boldsymbol{r}_1$ as a new model $\boldsymbol{r}1$ equipped with two projection mappings $\boldsymbol{r}_{11}$, $\boldsymbol{r}_{12}$, which are totally defined.

Thus, we have specified all our data via models and functional q-mappings as shown in the diagram below: arrows with hooked tails denote inclusions of models into their augmentations with derived elements computed with queries Qi.



2.2 Model merging: a sample multi-mapping scenario

We want to integrate data specified by the diagram above. We first need to merge models P_1, P_2 and P_{12} without data loss and duplication. The type discipline prescribes merging their metamodels first. To merge metamodels $\boldsymbol{\mathcal{P}}_1^+$, $\boldsymbol{\mathcal{P}}_2^+$, and $\boldsymbol{\mathcal{P}}_{12}^+$ (see Fig. 3), we take their disjoint union (no loss), and then glue together elements related by mappings $\boldsymbol{r}_{1,2}$ (to avoid duplication). The result is shown Fig. 5(a). There is a redundancy in the merge since attribute 'tname' and pair (title, lname) are mutually derivable. We need to choose either of them as a basic structure, then the other will be derived (see Fig. 5(b1,b2)) and could be omitted from the model. We call this process *normalization*. Thus, there are two normalized merged metamodels. Amongst the three metamodels to be merged, we favor metamodel $\boldsymbol{\mathcal{P}}_{12}$ in which attribute 'tname' is considered derived from 'title' and 'last name', and hence choose metamodel $\boldsymbol{\mathcal{P}}_{n1}^+$ as the merge result (below we omit the subindex).

Now we take the disjoint union of models P_1^+, P_2^+, P_{12}^+ (Fig. 3), and glue together elements linked by mappings $e_{1,2}$. Note that we merge attribute slots rather than values; naming conflicts are resolved in favor of names used in metamodel $\boldsymbol{\mathcal{P}}_{12}^+$. The merged model is shown in Fig. 6; the merged metamodel is clear

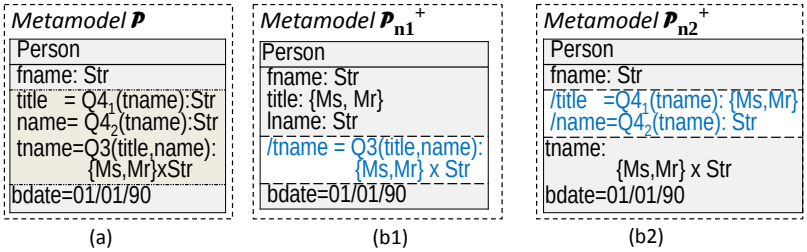


Fig. 5. Normalizing the merge

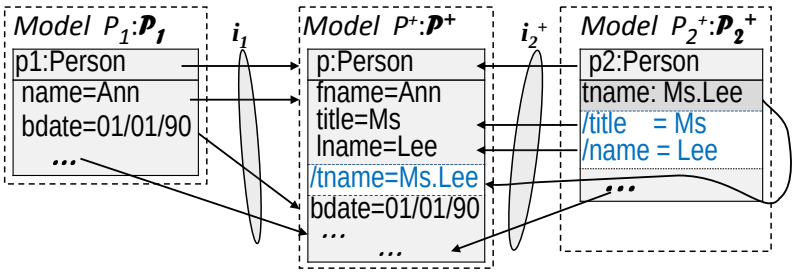


Fig. 6. Result of the merge modulo match in Fig. 3

and implicit. Note the interplay between basic-derived-element links in mapping e_2 in Fig. 3. Without them, the merge would contain redundancies. Note also that all three component models are embedded into the merge by injective mappings $i_{1,2,3}$ (mapping i_3 is evident and not shown).

Merge and integration, abstractly. The hexagon area in Fig. 7 presents the merge described above in an abstract way. Nodes in the diagram denote models, and arrows are functional mappings. Arrows with hooked tails are inclusions of models into their augmentations with derived elements computed with queries Qi. Computed mappings are shown with dashed arrows (blue if colored), and computed model P^+ is not framed.

However, building model P^+ does not finish integration. Our system of models also has two view models, S and A, and to complete integration, we need to show how views S and A are mapped into the merge P. For this goal, we need to translate queries Q_1 and Q_2 to, resp., models P_1 and P_2 from their original models to the merge model P^+ using mappings i_1, i_2. We first replace each element $x@P_k$ occurring in the expression defining query Q_k $(k = 1, 2)$ by the respective element $i_k(x)@P^+$. In this way query definitions are translated to model P^+. Then we execute them and augment model P^+ with the respective derived elements as shown by inclusion mappings $\eta_k^\sharp$ $(k = 1, 2)$ within the lane (a-b) in the figure. That is, we add to model P^+ derived attribute /age (on the left) and two derived subclasses, 'title=Ms' and 'title=Mr' (on the right). Since model P^+ is embedded into its augmentations $Q_k(P^+)$ $(k = 1, 2)$, and queries Q_k preserve

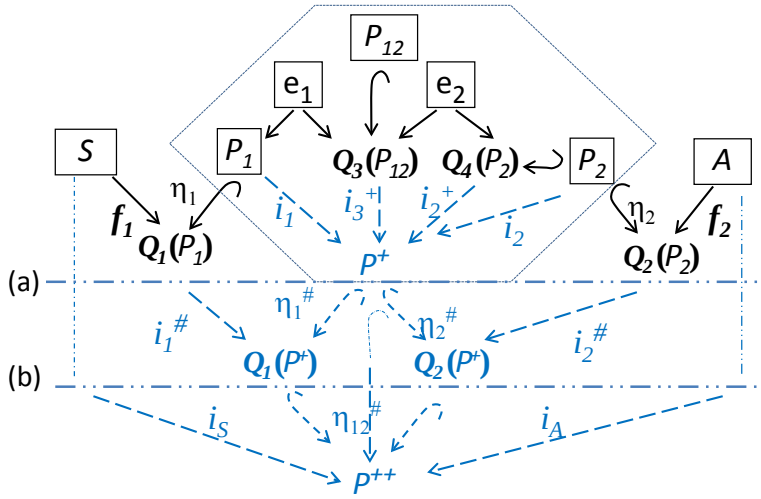


Fig. 7. The merge example, abstractly

data embedding (are *monotonic* in the database jargon), the result of executing Q_k against model P_k can be embedded into the result of executing Q_k against P^+. Hence, we have mappings $i_k^\sharp$ making the squares $[P_k\ P^+\ Q_k(P^+)\ Q_k(P_k)]$ $(k = 1, 2)$ commutative.

Finally, we merge queries Q_1 and Q_2 to model P^+ into query Q_{12}, whose execution adds to model P^+ both derived attribute /age and derived subclasses. We denote the resulting model by $P^+ +$ and η_{12}: $P^+ \hookrightarrow P^{++}$ is the corresponding inclusion (see the lower diamond in Fig. 7). Now we can complete integration by building mappings i_S: $S \to P^{++}$ and i_A: $A \to P^{++}$ by sequential composition of the respective components. These mappings say that Ms. Ann Lee is a student and an actor — information that is not provided by either of models P^+ or P^{++} as such.

2.3 The Kleisli construction

The diagram in Fig. 7 is precise but looks too detailed in comparison with the informal diagram Fig. 1(b). We want to design a more compact yet still precise notation for this diagram.

Note that the diagram heavily uses the following mapping pattern

$$X \xrightarrow{f} Q(Y) \xleftarrow{\eta} Y,$$

where X, Y are the source and the target models, $Q(Y)$ is augmentation of Y with elements computed by a query Q to Y, and η is the corresponding inclusion. The key idea of the Kleisli construction developed in category theory is to view this pattern as an arrow K: $X \Rightarrow Y$ comprising two components: a query Q_K to the target Y and a functional mapping f_K: $X \to Q_K(Y)$ into the corresponding

augmentation of the target. Thus, the query becomes a part of the mapping rather than of model Y, and we come to the notion of q-mapping mentioned above. We will often denote q-mappings by double-body arrows to recall that they encode both a query and a functional mapping. By a typical abuse of notation, a q-mapping and its second component (the functional mapping) will be often denoted by the same letter; we write, say, $f\colon X \Rightarrow Y$ and $f\colon X \to Q(Y)$ using letter f for both. With this notation, the input data for integration (framed nodes and solid arrows in diagram Fig. 7) are encoded by the following diagram

$$\boxed{S} \xRightarrow{f1} \boxed{P_1} \xleftarrow{e_1} \boxed{P_{12}} \xrightarrow{e_2} \boxed{P_2} \xLeftarrow{f_2} \boxed{A}$$

where spans e_1, e_2 from Fig. 7 are encoded by arrows with bullets in the middle. Note a nice similarity between this and our original diagram Fig. 1(b)(its upper row of arrows); however, in contrast to the latter, the arrows in the diagram above have the precise meaning of q-mappings.

Finally, we want to formalize the integration procedure as an instance of the colimit operation: as is well-known, the latter is a quite general pattern for "putting things together" [2]; see also [5, 12, 6] for concrete examples related to MDE. To realize the merge-as-colimit idea, we need to organize the universe of models and q-mappings into a category, that is, define identity q-mappings and composition of q-mappings. The former task is easy: given a model X, its identity q-mapping $\mathbb{1}_X : X \Rightarrow X$ is $1_X\colon X \to Q_\varnothing(X)$ where $Q_\varnothing$ is an empty query so that $Q_\varnothing(X) = X$, and 1_X is the identity mapping of X to itself.

Composition of q-mappings is, however, non-trivial. Given two composable q-mappings $f\colon X \Rightarrow Y$ and $g\colon Y \Rightarrow Z$, defining their composition $f;g\colon X \Rightarrow Z$ is not straightforward, as shown by the diagram in Fig. 8 (ignore the two dashed arrows and their target for a moment): indeed, after unraveling, mappings f and g are simply not composable. To manage the problem, we need to apply query Q_f to model $Q_g(Z)$ and correspondingly extend mapping g as shown in the diagram. Composition of two queries is again a query, and thus pair $(f; g^\#, Q_f \circ Q_g)$ determines a new q-mapping from X to Z.

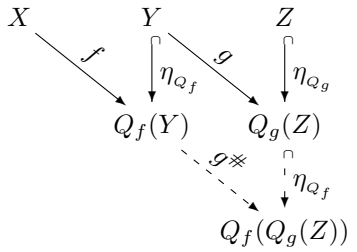


Fig. 8. Q-mapping composition

The passage from g to $g^\#$—the *Kleisli extension operation*—is crucial for the construction. (Note that we have used this operation in Fig. 7 too). On the level of metamodels and query definitions (syntax only), Kleisli extension is simple and amounts to term substitution. However, queries are executed for models, and an accurate formal definition of the Kleisli extension needs a certain amount of non-trivial work to be done. We outline the main points in the next two sections.

3 Model translation, traceability and fibrations

The carrier structure. We fix a category $\mathbb{G}$ with pullbacks, whose objects are to be thought of as graphs, or many-sorted (colored) graphs, or attributed graphs [13]. The key point is that they are definable by a metametamodel itself being a graph with, perhaps, a set of *equational* constraints. In precise categorical terms, we require $\mathbb{G}$ to be a presheaf topos [14], and hence possessing limits, colimits, and other important properties. In addition, it makes sense to say about objects' elements. We will call $\mathbb{G}$-objects *'graphs'*, and write $e \in G$ to say that e is an element of 'graph' G.

For a 'graph' M thought of as a metamodel, an *M-model* is a pair $A = (D_A, t_A)$ with D_A a 'graph' and $t_A\colon D_A \to M$ a mapping (arrow in category $\mathbb{G}$) to be thought of as *typing*. In a heterogeneous environment with models over different metamodels, we may say that a model A is merely an arrow $t_A\colon D_A \to M_A$ in $\mathbb{G}$, whose target M_A is called *the metamodel* of A, and source D_A is the *data carrier*. In our examples, a typing mapping for OIDs was set by colons: writing p:Person for a model A means that $p \in D_A$, Person $\in M_A$ and $t_A(p) =$ Person. For attributes, our notation covers even more, e.g., writing 'name=Ann' (nested in class Person) refers to some arrow $a\colon b \to$ Ann in 'graph' D_A, which is mapped by t_A to arrow *attrdom*: name $\to$ *string* in 'graph' M_A, but names of a and b are not essential for us. Details can be found in [12, Sect.3].

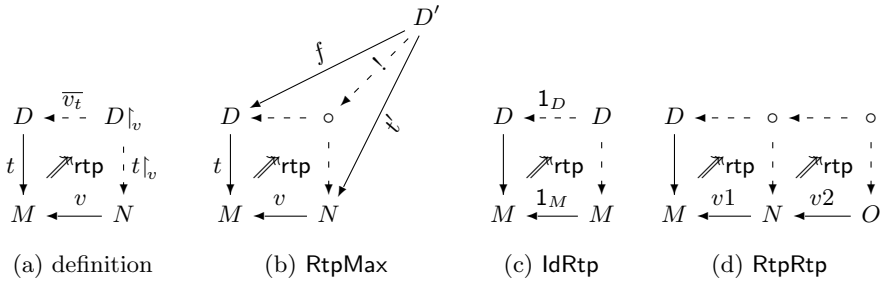
A *model mapping* $f\colon A \to B$ is a pair of $\mathbb{G}$-mappings, $f_{\text{meta}}\colon M_A \to M_B$ and $f_{\text{data}}\colon D_A \to D_B$, which commute with typing: $f_{\text{data}}; t_B = t_A; f_{\text{meta}}$. Below we will also write f_M for f_{meta} and f_D for f_{data}. Thus, a model mapping is actually a commutative diagram; we will usually draw typing mappings vertically and mappings f_M, f_D horizontally. We assume the latter to be monic (or injective) in $\mathbb{G}$ like in all our examples. This defines category ***Mod*** of models and (injective) model mappings.

As each model A is assigned with its metamodel M_A, and each model mapping $f\colon A \to B$ with its metamodel component $f_M\colon M_A \to M_B$, we have a projection mapping $\boldsymbol{p}\colon \boldsymbol{Mod} \to \boldsymbol{MMod}$ where we write ***MMod*** for category $\mathbb{G}$ or some its subcategory of 'graphs' that can serve as metamodels (e.g., all finite 'graphs'). It is easy to see that $\boldsymbol{p}$ preserves mapping composition and identities, and hence is a functor.

To take into account constraints, we need to consider metamodels as pairs $M = (G_M, C_M)$ with G_M a carrier 'graph' and C_M a set of constraints. Then not any typing $t_A\colon D_A \to G_M$ is a model: a legal t_A must also satisfy all constraints in C_M. Correspondingly, a legal mapping $f\colon M \to N$ must be a 'graph' mapping $G_M \to G_N$ compatible with constraints in a certain sense (see [12] for details). We do not formalize constraints in this paper, but in our abstract definitions below, objects of category ***MMod*** may be understood as pairs $M = (G_M, C_M)$ as above, and ***MMod***-arrows as legal metamodel mappings.

.

Retyping. Any metamodel mapping $v\colon M \leftarrow N$ generates retyping of models over M into models over N as shown by diagram on the right. If an element



(a) definition (b) RtpMax (c) IdRtp (d) RtpRtp

Fig. 9. Retyping operation: an application instance (a) and properties (laws) (b,c,d)

$e \in N$ is mapped to $v(e) \in M$, then any element in 'graph' D typed by $v(e)$, is retyped by e. Graph $D\restriction_v$ consists of so retyped elements of D, and mapping $\overline{v_t}$ traces their origin.

Formally, elements of $D\restriction_v$ can be identified with pairs $(e, d) \in N \times D$ such that $v(e) = t(d)$, and mappings $t\restriction_v$ and $\overline{v_t}$ are the respective projections. The operation just described is well-known in category theory by the name *pullback* (PB) : typing arrow $t\restriction_v$: $D\restriction_v \to N$ is obtained by *pulling back* arrow t along arrow v. If we want to emphasize the vertical dimension of the operation, we will say that traceability arrow $\overline{v_t}$ is obtained by *lifting* arrow v along t. We will also use a brief notation with names of derived models and mappings skipped as shown by the inner square in Fig. 9(b): derived arrows are dashed, and the derived node is blank.

If the type $t(d)$ of an element $d \in D$ is outside the range of mapping v, then d does not appear in $D\restriction_v$. Particularly, v being an inclusion, 'graph' $D\restriction_v$ is exactly the t-preimage of N, or its *inverse image* along t, and $\overline{v_t}$ is the respective inclusion. We will use this terminology for injective v's as well, and we assume that our metamodel mappings are injective.

Retyping has three remarkable algebraic properties specified in Fig. 9(b,c,d). Diagram (b) specifies *maximality* of the retyped 'graph' $D\restriction_v$ in the following sense. Any other 'graph' typed over N by mapping t' and mapped into D by f so that the outer square $DMND'$ commutes, can be uniquely mapped into $D\restriction_v$ (note the arrow '!') such that the two triangle diagrams commute. Maximality implies the uniqueness of the retyped graph up to isomorphism: any two retypings are isomorphic via the respective arrow !. (In category theory, this is the defining property of the pullback operation: a square is a pullback if it is maximal in the sense above.) Diagram (c) says that identical retyping does nothing: if v is identity, then $\overline{v_t}$ is identity as well. Diagram (d) reads as follows: if the two inner squares specify retyping, then the outer rectangle is also a retyping over composed mapping $v1; v2$; the corresponding label is skipped to ease the notation.

Abstract formulation via fibrations. Retyping can be specified as a special property of functor $\boldsymbol{p}$: $\boldsymbol{Mod} \to \boldsymbol{MMod}$. For an arrow $v\colon M \leftarrow N$ in $\boldsymbol{MMod}$, and

an object A over M (i.e., such that $\boldsymbol{p}(A) = M$), there is an arrow $\overline{v_A}\colon A \leftarrow A\restriction_v$ over v (i.e., a commutative diagram as shown in Fig. 9(a)) being maximal in the sense of diagram Fig. 9(b). Such an arrow is called the *(weak) $\boldsymbol{p}$-Cartesian lifting* of arrow v, and is defined up to canonical isomorphism. Functor $\boldsymbol{p}$ with a chosen Cartesian lifting for any arrow v, which satisfies the two properties (c,d) is called a *split fibration* (see, e.g., [15, Exercise 1.1.6]). Thus, the existence of model retyping over metamodel mappings can be abstractly described by saying that we have a split fibration $\boldsymbol{p}\colon \boldsymbol{Mod} \to \boldsymbol{MMod}$.

Definition 1 () An *(abstract) metamodeling framework* is a split fibration $\boldsymbol{p}\colon \boldsymbol{Mod} \to \boldsymbol{MMod}$ with the following terminology for its components. Elements of category ***Mod*** are called *models* and *model mappings*; elements of ***MMod*** are *metamodels* and *metamodel mappings*. For a model A and model mapping f, we write $A{:}M$ and $f{:}v$ if $\boldsymbol{p}(A) = M$ and $\boldsymbol{p}(f) = v$. Also, $\boldsymbol{p}$-Cartesian arrows in ***Mod*** are called *model retyping*. Given a metamodel mapping $v\colon M \leftarrow N$ and a model $A{:}M$, the corresponding retyping is denoted by $\overline{v_A}\colon A \leftarrow A\restriction_v$ (and is called Cartesian lifting, as usual for fibrations).

4 Query mechanism via monads

In this section we consider querying and its basic properties in abstract terms. In Section 4.1, we model querying algebraically as a diagram (tile) operation, and establish its basic equational laws. Sections 4.2-4.3 consider interaction of queries with model mappings also within the tile algebra framework; and new respective laws are introduced. Section 4.4 presents an accurate formalization based on monads.

4.1 Query as a tile operations and its basic properties.

Examples in Section 2 show that a query mechanism consists of two components: query definition and query execution. The former acts on the level of metamodels: a query definition Q_{def} against metamodel M can be seen as an inclusion $\eta_M^{Q_{\mathrm{def}}}\colon M \hookrightarrow M.Q_{\mathrm{def}}$ of M into its augmentation with derived elements.

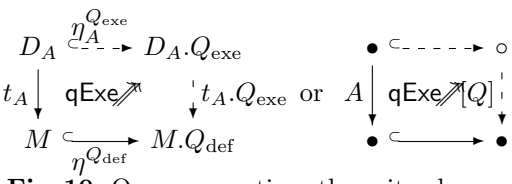
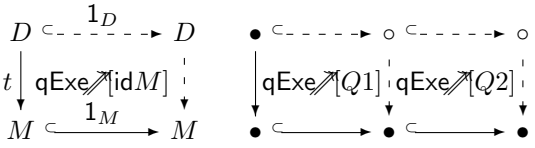


Fig. 10. Query execution: the arity shape

Execution of Q_{def} for an M-model A can be seen as an operation specified in Fig. 10 (left). Its input consists of two arrows, query definition $\eta^{Q_{\mathrm{def}}}$ and typing mapping t_A, and the output is the pair of dashed arrows: the upper one is inclusion of the carrier graph into its augmentation with the query results, and the vertical one is the corresponding typing. The diagram is commutative and specifies an inclusion $\eta_A^Q\colon A \hookrightarrow A.Q$ in category ***Mod***. Diagram Fig. 10 (right) presents a terse notation, in which the name of the query Q appears as a parameter in the operation label; the names of all other elements of the diagram can be restored.



(a) IdExe (b) ExeExe

Fig. 11. Query execution: laws

Any query mechanism has two remarkable algebraic properties specified in Fig. 11. Given a metamodel M, we postulate the existence of the identity query $\mathsf{id}M$ with $\eta^{\text{def}}_{\mathsf{id}M}$ being the identity $1_M\colon M \to M$, and diagram (a) says that execution of this query for any model D over M adds nothing to it. Diagram (b) says that composition of query executions, say, $\mathsf{exe}[Q1]$ and $\mathsf{exe}[Q2]$, amounts to the execution of the composed query $\eta^{(Q1;Q2)_{\text{def}}}\colon M \to M.Q1_{\text{def}}.Q2_{\text{def}}$, that is, the outer rectangle specifies execution of query $Q1;Q2$ (but the corresponding label is skipped to ease the notation).

We also require diagram Fig. 10 to be a pullback, i.e., data D_A is exactly the inverse image of M along $t_A.Q_{\text{exe}}$. This condition formalizes the fundamental requirement that querying (in contrast to updating) does not affect data: any item computed by a query has a new type.

4.2 Monotonicity.

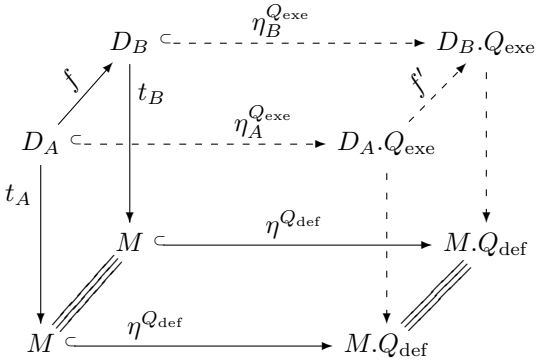


Fig. 12. Query monotonicity

Let M be a metamodel and Q_{def} is a query against it. Query Q_{def} is said to be *monotonic*, if it preserves dataset inclusion, to wit: given a model mapping $f\colon A \to B$ (i.e., an injection $f\colon A \to B$ commuting with typing as shown by the left face of the cube in Fig. 12), there is a model mapping $f.Q_{\text{exe}}\colon A.Q_{\text{exe}} \to B.Q_{\text{exe}}$, i.e., injection

$f'\colon D_A.Q_{\text{exe}} \to D_B.Q_{\text{exe}}$,

which makes the entire cube commutative.

It is well-know that a wide class of practically important relational queries — those without negation — are monotonic.

4.3 Query translation and locality.

Let Q be a query to model A as described by the front face of the cube in Fig. 13. Given a model mapping $f\colon A \to B$, i.e., a pair of injective mapping commuting with typing (see the right face of the cube and ignore the label $[\mathsf{pb}]$), we want to translate Q into a query Q^f against B, and compare the results of two executions.

Query definition translation is easy: we only replace each M_A's element e occurring into Q_{def} by M_B's element $f(e)$; this gives us mapping f'_M: $M_A.Q_{\mathrm{def}} \to M_B.Q^f_{\mathrm{exe}}$ and commutative bottom square of the cube. In fact, this is nothing but syntactical substitution (note the label subst on the bottom face). We may say that query Q_{def} is applied to metamodel mappings too, and write $f.Q_{\mathrm{def}}$ for f'_M. Evidently, given two consecutive mappings f: $M \to N$, g: $N \to O$ between metamodels, we have $(f;g).Q_{\mathrm{def}} = (f.Q_{\mathrm{def}});(g.Q_{\mathrm{def}})$. Also, $1_M.Q_{\mathrm{def}} = 1_M$.

Interrelation between executions, $D_A.Q^f_{\mathrm{exe}}$ and $D_B.Q^f_{\mathrm{exe}}$, is much more complicated because query execution is about real operations with data (in contrast to purely syntactical query definition). We will first consider two particular cases. The first one is when model A is the inverse image of model B along mapping f_M, i.e., any D_B-element whose type belongs to $f_M(M_A)$ is carried to D_A, and so the left face of the cube is a pullback. In this case, execution of query Q^f against D_B amounts to executing Q over D_A up to OID-renaming via f_D, and hence $D_A.Q_{\mathrm{exe}}$ is an inverse image of $D_B.Q^f_{\mathrm{exe}}$ along mapping f'_M. In other words, if the left face is a pullback, and the bottom face is the query definition translation (substitution), then query execution makes a commutative cube as shown in Fig. 13, and the right face is a pullback too.

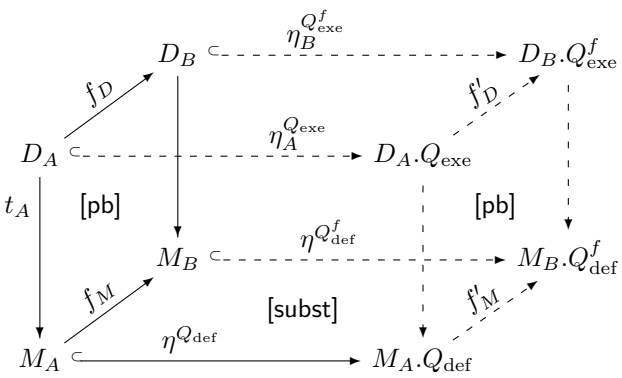


Fig. 13. Query translation and locality

A default assumption in the above arguments is that D_B-elements whose types are beyond $f_M(M_A)$ do not affect execution of Q^f. We may consider the latter condition as *locality* of query execution, and typical queries are surely local in this sense. Below we will assume locality by default. Thus, query execution translates pulbbacks to pullbacks.

The second special case we consider is somewhat the opposite: now we assume that two models are over the same metamodel, $M_A = M_B = M$, but the data 'graph' D_A is embedded into graph of D_B. This case is specified by the left face of cube Fig. 12, and we can apply monotonicity law. (Note that this face can only be a pullback in the trivial case of A and B being the same up to isomorphic OID renaming; in all other cases D_A can be considered as a proper 'subgraph' of D_B.)

Now we note that any heterogeneous model mapping $f = (f_M, f_D)$: $A \to B$ can be factorized into a homogeneous mapping into the inverse image of D_B along f_M as shown in Fig. 9(b). Hence, conditions specified by Fig. 12 and

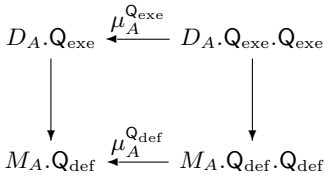


Fig. 14.

Fig. 13 together provide existence of mapping $f.Q_{\mathrm{exe}}$: $D_A.Q_{\mathrm{exe}} \to D_B.Q_{\mathrm{exe}}$ for a local monotonic query Q. Thus, a query Q gives rise to a graph morphism $[Q]$: $[\boldsymbol{Mod}] \to [\boldsymbol{Mod}]$, where $[\boldsymbol{Mod}]$ is the underlying graph of category $\boldsymbol{Mod}$.

It is a standard categorical exercise in diagram chasing to see that for two consecutive model mappings f: $A \to B$, g: $B \to C$, we have $(f;g).Q_{\mathrm{exe}} = (f.Q_{\mathrm{exe}});(g.Q_{\mathrm{exe}})$ (assuming that query Q is monotonic and local). Also, $\mathbf{1}_A.Q_{\mathrm{exe}} = \mathbf{1}_{A.Q_{\mathrm{exe}}}$.

4.4 Query mechanism via monads and fibrations

Query languages are monads. So far, given a query language QL over a metamodeling framework $\boldsymbol{p}$: $\boldsymbol{Mod} \to \boldsymbol{MMod}$, we have been considering individual QL-queries η_A^Q: $A \hookrightarrow A.Q$ (consisting of definition and execution as shown in Fig. 11(a)). It is technically convenient to merge all these into one huge "query" η_A^{Q}: $A \hookrightarrow A.\mathsf{Q}$, where $A.\mathsf{Q}$ denotes model A augmented with all possible derived elements computed by all possible QL-queries to A. Our discussion above shows that we need to require mapping Q to act also on model mappings and, moreover, to be a functor Q: $\boldsymbol{Mod} \to \boldsymbol{Mod}$. Moreover, if mapping f: $A \to B$ is a pullback square, then mapping $f.\mathsf{Q}$: $A.\mathsf{Q} \to B.\mathsf{Q}$ is a pullback square as well (recall locality laws in Section 4.3).

Sequential query composition Fig. 11(b) gives rise to a mapping μ_A^{Q}: $A.\mathsf{Q}.\mathsf{Q} \to A.\mathsf{Q}$. Indeed, if $Q1$ is a query to A and $Q2$ is a query to $A.Q_1$, then each derived element in $A.Q_1.Q_2$ is actually an element of $A.(Q1;Q2) \subset A.\mathsf{Q}$, hence the mapping μ_A^{Q} above. In more detail, this mapping is the commutative diagram in Fig. 14. Note also all elements in model $A.\mathsf{Q}_{\mathrm{exe}}.\mathsf{Q}_{\mathrm{exe}}$ are, in fact, already computed in $A.\mathsf{Q}_{\mathrm{exe}}$ and thus are copied from $A.\mathsf{Q}_{\mathrm{exe}}$ to $A.\mathsf{Q}_{\mathrm{exe}}.\mathsf{Q}_{\mathrm{exe}}$ and correspondingly retyped according to mapping μ_A^{def}. That is, the diagram we discuss is a pullback.

Thus, a monotonic query language gives rise to a triple $(\mathsf{Q}, \eta^{\mathsf{Q}}, \mu^{\mathsf{Q}})$ with Q an endofunctor on category $\boldsymbol{Mod}$, and η^{Q}, μ^{Q} two mappings that assign to any model $A \in \boldsymbol{Mod}$ model mappings η_M^{Q} and μ_A^{Q} as above. It is straightforward to check that we should also require commutativity of three (standard for categorical algebra) diagrams in Fig. 15.

A triple $(\mathsf{Q}, \eta^{\mathsf{Q}}, \mu^{\mathsf{Q}})$ satisfying the three commutativity conditions is called a *monad*, and so we define a monotonic query language as a monad over $\boldsymbol{Mod}$.

Importantly, constituting mappings of this monad have special properties related to pullbacks. To wit: for any model A, mappings η_A^{Q}: $A \to A.\mathsf{Q}$ and

μ^{Q}_A: $A.\mathsf{Q}.\mathsf{Q} \to A.\mathsf{Q}$ are pullback squares (the former is due to the requirement of data preservation, and the latter is just discussed). Locality condition discussed in Section 4.3 means that functor Q preserves pullback squares. Three conditions above mean that the monad Q is Cartesian.

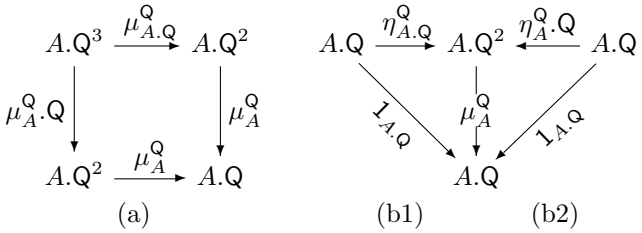


Fig. 15. Laws for a query monad

Query execution vs. query definition. A fundamental feature of querying (in contrast to updating) is that query execution always "run" over the respective query definition (Section 4.1). We formalize this property by (a) postulating a query definition monad $(\mathsf{Q}_{\mathrm{def}}, \eta^{\mathsf{Q}_{\mathrm{def}}}, \mu^{\mathsf{Q}_{\mathrm{def}}})$ over the category of metamodels ***MMod***, and (b) requiring that projection functor $\boldsymbol{p}$: ***Mod*** $\to$ ***MMod*** be a monad morphism, that is, for any model A the following three conditions hold: (b1) $A.\mathsf{Q}.\boldsymbol{p} = A.\boldsymbol{p}.\mathsf{Q}_{\mathrm{def}}$, (b2) $\eta^{\mathsf{Q}}_A.\boldsymbol{p} = \eta^{\mathsf{Q}_{\mathrm{def}}}_A$, and (b3) $\mu^{\mathsf{Q}}_A.\boldsymbol{p} = \mu^{\mathsf{Q}_{\mathrm{def}}}_A$ for any model A.

Recall that projection functor $\boldsymbol{p}$ is actually a (retyping) fibration as discussed at the end of Sect. 3. In these terms, the three special pullback properties of the query monad mean that mappings η^{Q}_A and μ^{Q}_A are $\boldsymbol{p}$-Cartesian for any model A, and functor Q preserves Cartesianity. We will thus call monad Q $\boldsymbol{p}$-*Cartesian*. We summarize the discussion by the following main definition.

Definition 2 (main) A *monotonic query language* over an abstract metamodeling framework $\boldsymbol{p}$: ***Mod*** $\to$ ***MMod*** is a pair of monads $(\mathsf{Q}, \mathsf{Q}_{\mathrm{def}})$ over categories ***Mod*** and ***MMod*** resp. such that $\boldsymbol{p}$ is a monad morphism (conditions (b1-b3) above), and monad Q is $\boldsymbol{p}$-Cartesian, i.e., functor Q preserves $\boldsymbol{p}$-Cartesianity, and mappings η^{Q}_A, μ^{Q}_A are $\boldsymbol{p}$-Cartesian for any model A.

5 View mechanism via Kleisli construction

Background. A monad Q over a category $\boldsymbol{C}$ generates its *Kleisli* category $\boldsymbol{C}_{\mathsf{Q}}$ as follows. It has the same objects as $\boldsymbol{C}$, but a $\boldsymbol{C}_{\mathsf{Q}}$-arrow $f: Y \Rightarrow X$ is a $\boldsymbol{C}$-arrow $f: Y \to X.\mathsf{Q}$. Composition of $\boldsymbol{C}_{\mathsf{Q}}$-arrows, say, $g: Z \to Y.\mathsf{Q}$ and $f: Y \to X.\mathsf{Q}$, is not immediate since f's source and g's target do not match. It is defined as shown in Fig. 16: $g;f: Z \Rightarrow X$ in $\boldsymbol{C}_{\mathsf{Q}}$ is $g;f^{\#}: Z \to X.\mathsf{Q}$ in $\boldsymbol{C}$, where $f^{\sharp} = f.\mathsf{Q};\mu^{\mathsf{Q}}_{Y.\mathsf{Q}}$. The $\boldsymbol{C}_{\mathsf{Q}}$-identity loop of object X, $1_X: X \Rightarrow X$ is $\boldsymbol{C}$-arrow $\eta^{\mathsf{Q}}_X: X \to X.\mathsf{Q}$. Monad axioms guarantee associativity of composition and unitality of identity, thus, $\boldsymbol{C}_{\mathsf{Q}}$ is indeed a category.

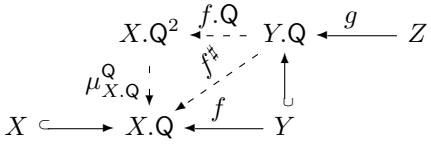


Fig. 16. The Kleisli construction

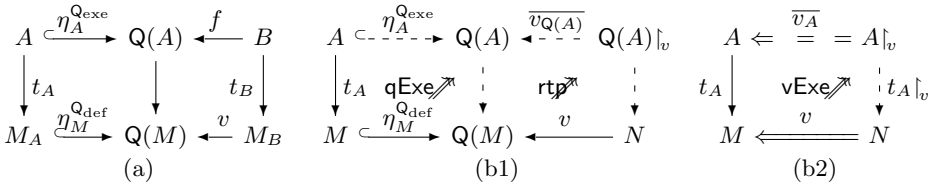


Fig. 17. Q-mappings (a) and view mechanism (b1,b2)

Lemma 1 ([16]). *If category* $\boldsymbol{C}$ *has colimits of all diagram from a certain class* $\mathcal{D}$*, then the Kleisli category* $\boldsymbol{C}_{\mathsf{Q}}$ *has* $\mathcal{D}$*-colimits as well.*

Kleisli for query monads. In terms of a metamodeling framework, the Kleisli construction has an immediate practical interpretation. Let $(\mathsf{Q}, \mathsf{Q}_{\mathrm{def}})$ be a monotonic query language over a metamodeling framework $\boldsymbol{p}$: $\boldsymbol{Mod} \to \boldsymbol{MMod}$. Arrows in the Kleisli category $\boldsymbol{Mod}_{\mathsf{Q}}$ are shown in Fig. 17(a). They are, in fact, the q-mappings we considered in our examples, and we will also denote category $\boldsymbol{Mod}_{\mathsf{Q}}$ by $\boldsymbol{qMap}_{\mathsf{Q}}$ (we thus switch attention from objects of the category to its arrows). It immediately allows us to state (based on Lemma 1) that if $\mathcal{D}$-shaped configurations of models related by ordinary (not q-) model mappings are mergeable, then $\mathcal{D}$-shaped configurations of models and q-mappings are mergeable as well. For example, merge in our running example can be specified as the colimit of the chain of models and Kleilsi mappings in Fig. **??**

Metamodel-level components of q-mappings between models are arrows in $\boldsymbol{MMod}_{\mathsf{Q}_{\mathrm{def}}}$, and they are nothing but view definitions: they map elements of the source metamodel to queries against the target one. Hence, we denote $\boldsymbol{MMod}_{\mathsf{Q}_{\mathrm{def}}}$ by $\boldsymbol{viewDef}_{\mathsf{Q}_{\mathrm{def}}}$ (and Lemma 1 is applicable again). View definitions can be executed as shown in Fig. 17(b1): first the query is executed, and then the result is retyped along the mapping v (recall that dashed arrows denote derived mappings).

The resulting operation of *view execution* is specified in Fig. 17(b2), where double arrows denote Kleisli mappings.Properties of the view execution mechanism are specified by Theorem 1.

Theorem 1. *Let* $(\mathsf{Q}, \mathsf{Q}_{\mathrm{def}})$ *be a monotonic query language over an abstract metamodeling framework* $\boldsymbol{p}$: $\boldsymbol{Mod} \to \boldsymbol{MMod}$*. It gives rise to a split fibration* $\boldsymbol{p}_{\mathsf{Q}}$: $\boldsymbol{qMap}_{\mathsf{Q}} \to \boldsymbol{viewDef}_{\mathsf{Q}_{\mathrm{def}}}$ *between the corresponding Kleisli categories (and hence the view execution mechanism satisfies the laws in Fig. 18(a,b)).*

Proof. Functoriality of projection mapping $\boldsymbol{p}_{\mathsf{Q}}$ is evident. Maximality property of retyping Fig. 9(b) provides similar maximality of view execution. Thus, view execution arrows in Fig. 17(b2) (residing in $\boldsymbol{qMap}_{\mathsf{Q}}$) are $\boldsymbol{p}_{\mathsf{Q}}$-Cartesian, which defines the lifting operation. Law Fig. 18(a) is implied by the definition of identity loops in $\boldsymbol{qMap}_{\mathsf{Q}}$(which are inclusions η^{Q}_A in $\boldsymbol{Mod}$) and law Fig. 11(a). To prove Fig. 18(b), consider sequential Kleisli composition defined in Fig. 16. If f is Cartesian (prefix $\boldsymbol{p}$ is skipped), then $f.\mathsf{Q}$ is also Cartesian by Cartesianity of Q. Arrows μ^{Q}_A are always Cartesian, and hence arrow $f^\sharp = f.\mathsf{Q}; \mu^{\mathsf{Q}}_A$ is also Cartesian because $\boldsymbol{p}$ is fibration. Hence, if g and f are both Cartesian, then

composition $g; f^\sharp$ is Cartesian as well. But this composition in $\boldsymbol{Mod}$ is nothing but composition in the Kleisli category $\boldsymbol{qMap}_{\mathsf{Q}} = \boldsymbol{Mod}_{\mathsf{Q}}$. □

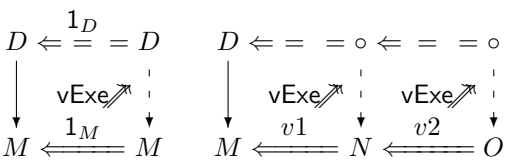


Fig. 18.

Law Fig. 18(b) says that execution of a sequentially composed view amounts to sequential composition of executions of the components. It is an evident requirement to a reasonable view mechanism; Theorem 1 shows that implementing view computation via querying followed by retyping does ensure this property, and that our formal model works well.

6 Related work

Modeling inductively generated syntactic structures (term and formula algebras) by monads and Kleisli categories is well known, e.g., [17, 10]. Semantic structures (algebras) then appear as Eilenberg-Moore algebras of the monad. In our approach, carriers of algebraic operations stay within the Kleilsi category. It only works for monotonic query languages, but the latter form a large, practically interesting class. (E.g, it is known that Select-Project-Join queries are monotonic.) We are not aware of a similar treatment of query languages in the literature.

Our notion of metamodeling framework is close to *specification frames* in institution theory [18]. Indeed, inverting the projection functor gives us a functor $\boldsymbol{p}_{\mathsf{Q}}^{-1}$: $\boldsymbol{viewDef}^{op}_{\mathsf{Q}_{\mathrm{def}}} \to \mathbf{Cat}$, which may be interpreted in institutional terms as mapping theories into their categories of models, and theory mappings into translation functors. The picture still lacks constraints, but adding them is not too difficult and can be found in [19]. Conversely, there are attempts to add query facilities to institutions via so called *parchments* [20]. Semantics in these attempts is modeled in a far more complex way than in our approach.

In several papers, Guerra et al. developed a systematic approach to intermodeling based on TGG (Triple Graph Grammars), see [1] for references. The query mechanism is somehow encoded in TGG-production rules, but precise relationships between this and our approach remain to be elucidated.

Our paper [7] heavily uses view definitions and views in the context of defining consistency for heterogeneous multimodels, and is actually based on constructs similar to our metamodeling framework. However, the examples therein go one step "down" in the MOF-metamodeling hierarchy in comparison with our examples, and formalization is not provided. The combination of those structures with structures in our paper makes a two-level metamodeling framework (a fibration over a fibration); studying this structure is left for future work.

7 Conclusion

The central notion of the paper is that of a q-mapping, which maps elements in the source model to queries applied to the target model. We have shown that

q-mappings provide a concise and clear specification framework for intermodeling scenarios, particularly, model merge. Composition of q-mappings is not straightforward: it requires free term substitution on the level of query definition (syntax), and actual operation composition on the level of query execution (semantics). To manage the problem, we model both syntax and semantics of a monotonic query language by a Cartesian monad over the fibration of models over their metamodels. Then q-mappings become Kleilsi mappings of the monad, and can be respectively composed. In this way the universe of models and q-mappings gives rise to a category (the Kleisli category of the monad), which provides a manageable algebraic foundation for specifying intermodeling scenarios.

References

1. Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F.: Inter-modelling: From theory to practice. In: MoDELS. Volume 6394 of LNCS., Springer (2010) 376–391
2. Goguen, J.: A categorical manifesto. Mathematical structures in computer science **1**(1) (1991) 49–67
3. José Fiadeiro: Categories for Software Engineering. Springer (2004)
4. Batory, D.S., Azanza, M., Saraiva, J.: The objects and arrows of computational design. In: MoDELS. Volume 5301 of LNCS., Springer (2008) 1–20
5. Sabetzadeh, M., Easterbrook, S.M.: View merging in the presence of incompleteness and inconsistency. Requir. Eng. **11**(3) (2006) 174–193
6. Rossini, A., Rutle, A., Lamo, Y., Wolter, U.: A formalisation of the copy-modify-merge approach to version control in mde. J. Log. Algebr. Program. **79**(7) (2010) 636–658
7. Diskin, Z., Xiong, Y., Czarnecki, K.: Specifying overlaps of heterogeneous models for global consistency checking. In: MoDELS Workshops. Volume 6627 of Lecture Notes in Computer Science., Springer (2010) 165–179
8. Romero, J., Jaen, J., Vallecillo, A.: Realizing correspondences in multi-viewpoint specifications. In: EDOC, IEEE Computer Society (2009) 163–172
9. Bernstein, P.: Applying model management to classical metadata problems. In: Proc. CIDR'2003. (2003) 209–220
10. Moggi, E.: Notions of computation and monads. Information and Computation **93**(1) (1991) 55–92
11. Diskin, Z., Maibaum, T., Czarnecki, K.: Intermodeling, queries, and Kleisli categories. Technical Report GSDLab-TR 2011-10-01, University of Waterloo (2011) http://gsd.uwaterloo.ca/QMapTR .
12. Diskin, Z.: Model synchronization: mappings, tile algebra, and categories. In R. Lämmel et al., ed.: Postproceedings GTTSE 2009. LNCS#6491, Springer (2011)
13. Ehrig, H., Ehrig, K., Prange, U., Taenzer, G.: Fundamentals of Algebraic Graph Transformation. (2006)
14. Barr, M., Wells, C.: Category theory for computing science. PrenticeHall (1995)
15. Jacobs, B.: Categorical logic and type theory. Elsevier Science Publishers (1999)
16. Manes, E.: Algebraic Theories. Springer (1976)
17. Jüllig, R., Srinivas, Y.V., Liu, J.: Specware: An advanced evironment for the formal development of complex software systems. In: AMAST. Volume 1101 of Lecture Notes in Computer Science., Springer (1996) 551–554

18. Goguen, J., Burstall, R.: Institutions: Abstract model theory for specification and programming. Journal of ACM **39**(1) (1992) 95–146
19. Diskin, Z.: Towards generic formal semantics for consistency of heterogeneous multimodels. Technical Report GSDLAB 2011-02-01, University of Waterloo (2011)
20. J.Goguen, Burstall, R.: A study in the foundations of programming methodology: Specifications, institutions, charters and parchments. Volume 240 of Springer Lect.Notes in Comp.Sci. (1986) 313–333