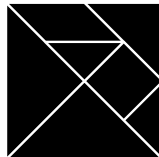
GSDLAB TECHNICAL REPORT

Intermodeling, queries and Kleisli categories

Zinovy Diskin, Tom Maibaum, and Krzysztof Czarnecki

GSDLAB–TR 2011–10–01 October 2011



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

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

The GSDLAB technical reports are published as a means to ensure timely dissemination of scholarly and technical work on a non-commercial basis. Copyright and all rights therein are maintained by the authors or by other copyright holders, notwithstanding that they have offered their works here electronically. It is understood that all persons copying this information will adhere to the terms and constraints invoked by each author's copyright. These works may not be reposted without the explicit permission of the copyright holder.

Intermodeling, queries and Kleisli categories

Zinovy Diskin[1,2], Tom Maibaum[1], and Krzysztof Czarnecki[2]

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

zdiskin@gsd.uwaterloo.ca,tom@maibaum.org,kczarnec@gsd.uwaterloo.ca

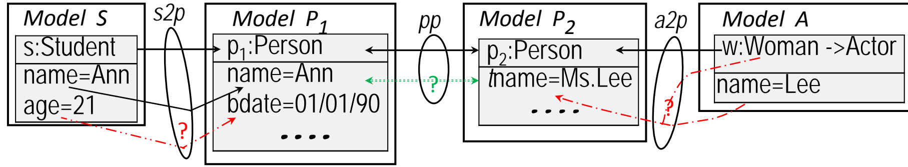
Abstract. Specification and maintenance of relationships between models are vital for MDE. We show that a wide class of such relationships can be specified in a compact and precise manner if inter-model mappings involve derived model elements computed by corresponding queries. Composition of such mappings 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 and formalized.

1 Introduction

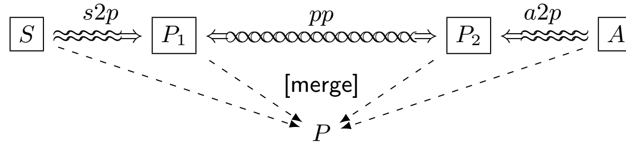
Model-driven engineering (MDE) is a prominent approach to software development, in which models of the domain and the software system are primary assets of the development process. Normally models are inter-related, perhaps in a very complex way, and to keep them consistent and use them coherently, relationships 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 four simple models in Fig. 1(a). 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. Class Woman extends class Person in model A, and ellipses in models P_1, P_2 refer to other attributes not shown. 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. Model P_1 gives the first name; P_2 provides the last name and the title of the person ('tname'). In Fig. 1(b), we show informally this relationship by a fancy arrow pp "intermapping" the two 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



(a) four models linked informally



(b) schema of the system

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

(in the database jargon, S is a materialized view of P_1). Model A plays a similar role for P_2. In Fig. 1(b), informal arrows $s2p$ and $a2p$ "map" views to their sources. The relationships 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 of 1990. Dashed arrows in diagram Fig. 1(b) informally denote mappings that embed the models into the merge.

We argue that building model management tools capable to perform integration like above for industrial models (normally containing thousands elements) requires clear and precise specifications of intermodel relationships. We are looking for a theoretical 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 a description to be generic and applicable to a wide class of metamodels. Category theory provides a suitable methodological framework (cf. [2, 3]) (e.g., merge can be defined as the colimit of the corresponding diagram [4]), but 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 extensionally, a mapping should be a set of links between models' elements as informally 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 (black with a color display); 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 the former is "a part" of the latter—note the question mark over the left dash-dotted (red) link. Even more complex is the relationship between attribute 'tname'@P_2 (name with title) and the Woman-Actor subclassing in model A, which is informally shown by a two-to-one dash-dotted link. Finally,

the dotted (green) link between elements name@P_1 and tname@P_2 encodes a great deal of semantic information described above. In the literature, such indirect relationships are usually specified by *correspondence rules* [5] or *expressions* [6] attached to the respective links. When such annotated links are composed, it is not clear how to compose the rules — the importance and difficulty of this problem was stressed in [6].

The paper makes the following three contributions. First, we show that for a wide class of inter-model 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 in the target model, that is, to be computable by corresponding queries. The multi-ary links are thus eliminated and replaced by binary links targeting at elements derived with multi-ary queries. Second, we demonstrate that such query-based mappings are instrumental for specifying and guiding model merge. Third, we show that the query mechanism can be mathematically modeled by a closure operator acting on models, in categorical terms, a *monad*. We state several important properties of monads resulting from query languages, and give them a compact algebraic characterization in terms of a categorical construct called *fibration*. Importantly, as soon as the query mechanism is specified by a monad, mappings involving derived elements can be formalized as *Kleisli morphisms* of the monad; they are known to be composable and constitute the *Kleisli category* generated by the monad. Thus, the universe of models and inter-model mappings can be structured as a category, and so operations over models and mappings can be specified by standard algebraic means developed in category theory.

The structure of the paper is as follows. In Section 2 we consider our running example in more detail, and show how derived elements allow us to resolve the issues mentioned above. Section 3 formalizes the basics: models' conformance to metamodels and retyping, and explains them in terms of fibrations. Section 4 formalizes query mechanism in terms of monads and fibrations. Section 5 describes related work and Section 6 presents our conclusion. Due to space limitations, several important discussions and mathematical explanations are omitted, but can be found in our technical report [7] 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 the running example to demonstrate important aspects of intermodeling to be addressed in our specification framework. We summarize the analysis with a list of requirements to be addressed by the formal framework.

2.1 From informal to formal mappings

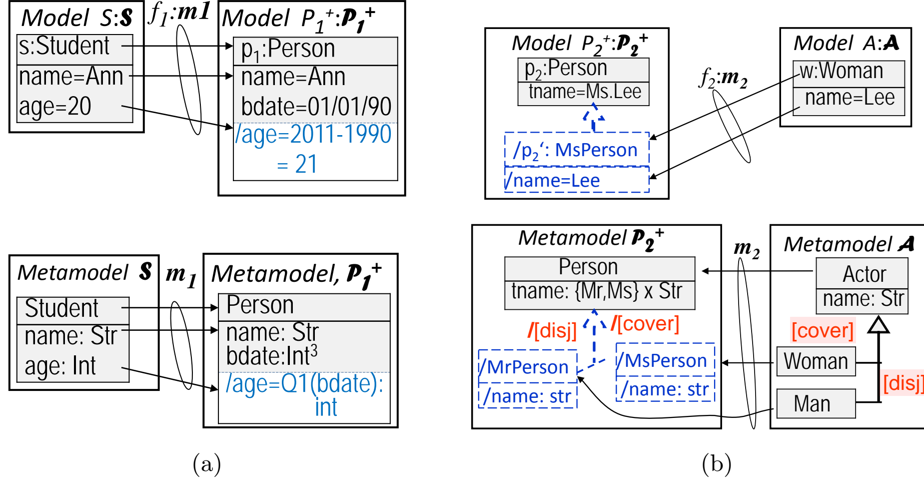
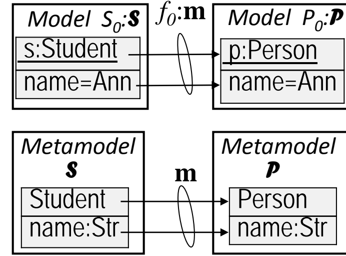


Fig. 2. Indirect matching via queries and direct mappings



Type discipline. Before mapping models we need to map their metamodels. Suppose we need to match models S_0 and P_0 over metamodels $\boldsymbol{\mathcal{S}}$ and $\boldsymbol{\mathcal{P}}$ resp. (see 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 related. Indeed, the two objects can only be matched if we know that their types actually refer to the same, or, at least, not disjoint, classes of real world objects. To make this knowledge explicit in our specification, we need to match the metamodels $\boldsymbol{\mathcal{S}}$ and $\boldsymbol{\mathcal{P}}$ via mapping $\boldsymbol{m}$. After that we can type-safely link 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. Indirectly related elements (dash-dotted links in Fig. 1(a)) can be specified by direct links to derived elements. For example, 'age' can be derived from 'bdate' with a suitable query—we write /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 term $2011 - \text{bdate.byear}$ is its body (definition); 'byear' denotes the year-field of the bdate-records. Now we can directly map metamodel $\boldsymbol{\mathcal{S}}$ to metamodel $\boldsymbol{\mathcal{P}}_1^+$ extending $\boldsymbol{\mathcal{P}}_1$ with the derived attribute as shown in Fig. 2(a). Query $Q1$ can be executed for any model over metamodel $\boldsymbol{\mathcal{P}}_1$, in particular, P_1, 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), where basic elements are shaded whereas derived elements are blank (and their names are prefixed with slash). The same idea works for

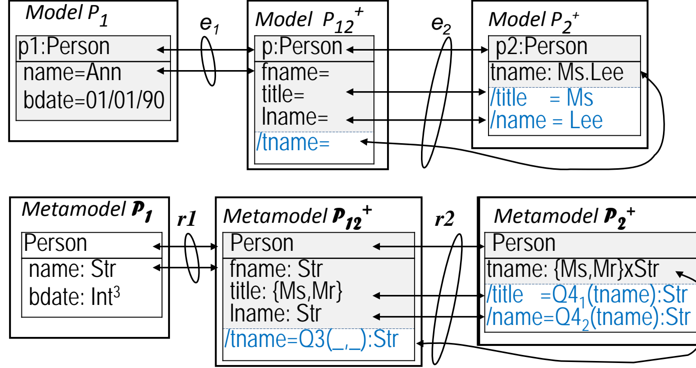


Fig. 3. Matching via spans and queries

the second dashed-dotted link in Fig. 1(a) (between models P_2 and A). The only difference is that now derived elements are computed by a more complex query (with two *select-from-where* clauses); the result is shown in Fig. 2(b). (Ignore for a moment (red) constraints in square brackets.) In this way we formalize informal mappings $s2p$, $a2p$ in Fig. 1(a) by formal mappings into models and metamodels augmented with derived elements. We will call such mappings *q-mappings* with prefix 'q' referring to 'using queries'. In category theory such mappings are called *Kleisli morphisms* or *Kleisli mappings*. Note that ordinary mappings can be seen as degenerate q-mappings that do not use derived elements.

Queries and constraints. Queries not only compute new (derived) elements, but also derive new constraints to be thought of as postconditions. For example, two derived subclasses of class Student in model P_2 are disjoint and cover the superclass (because set {Ms,Mr} is the disjoint union of singletons {Ms} and {Mr}). Thus, the query (say, $Q2$) which computes the subclasses also provides two derived constraints shown in metamodel $\boldsymbol{\mathcal{P}}_2^+$ in square brackets (and slashed). Note that mapping $\boldsymbol{m_2}$ is compatible with constraints declared in the metamodels: it translates a partition in $\boldsymbol{\mathcal{A}}$ into a partition in $\boldsymbol{\mathcal{P}}_2^+$.

Links-with-new-data via spans. We consider dotted (green) link p_1–p_2 in Fig. 1, whose semantics was informally explained in the Introduction. A much more precise description is given by Fig. 3. We first introduce a new metamodel $\boldsymbol{\mathcal{P}}_{12}$ that specifies new concepts assumed by the semantics. Then we relate these new concepts to the original ones via mappings $\boldsymbol{r}_1$, $\boldsymbol{r}_2$. We also show that attribute 'tname' in metamodel $\boldsymbol{\mathcal{P}}_2$ is nothing but the derived attribute $/\text{tname} = Q_3(\text{title}, \text{lname})$ in $\boldsymbol{\mathcal{P}}_{12}^+$, where query Q_3 is the pairing operation. We also introduce a new model P_{12} to declare sameness of objects $p_1@P_1$ and $p_2@P_2$, and to relate attribute slots in models P_{12} and P_1, P_2^+. The new attribute slots are kept empty — they will be filled-in with the corresponding local values during the merge.

It is well-known that an algebra for managing totally defined functions is much simpler than for 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

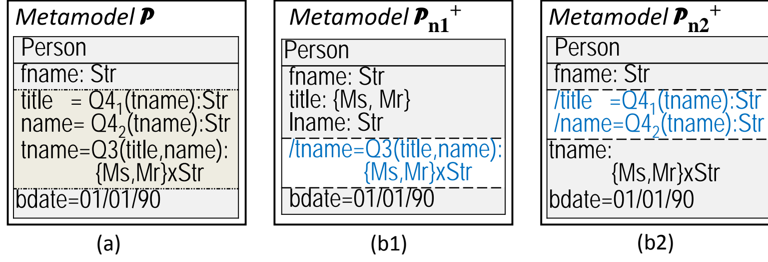


Fig. 5. Normalizing the merge

in our diagrams). To specify 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.

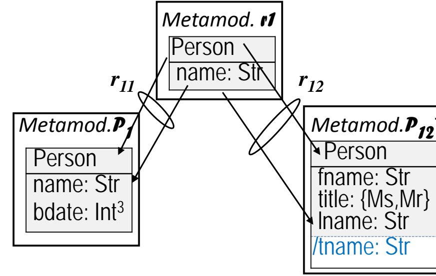


Fig. 4. Partial mappings via spans

Thus, we have specified all our data via models and q-mappings as shown in Fig. 7 above line (a) — ignore dashed (blue) arrows and non-framed (blue) model P^+, which are produced by the integration procedure considered below; arrows with hooked tails denote inclusions of models into their augmentations with derived elements computed with queries Qi.

2.2 Model merging as a sample multi-mapping scenario

We want to integrate data specified in the upper part of the hexagon in Fig. 7. We first need to merge models P_1, P_2 and P_{12} without data loss and duplication. The type discipline prescribes merge first metamodels and then models. To merge metamodels $\boldsymbol{\mathcal{P}}_1^+$, $\boldsymbol{\mathcal{P}}_2^+$, and $\boldsymbol{\mathcal{P}}_{12}^+$ 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 still some 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. We favor model $\boldsymbol{\mathcal{P}}_{12}$ in which attribute 'tname' is considered derived from 'title' and 'last name', and 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}^+, and glue together elements linked by mappings $e_{1,2}$. Note that we merge attribute slots but not values; naming conflicts are resolved in favor of names used in metamodel $\boldsymbol{\mathcal{P}}_{12}^+$.

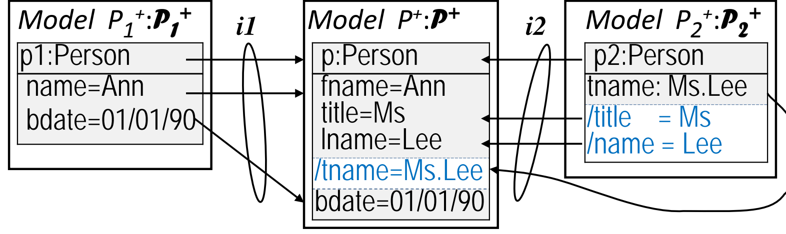


Fig. 6. Result of the merge modulo match in Fig. 3

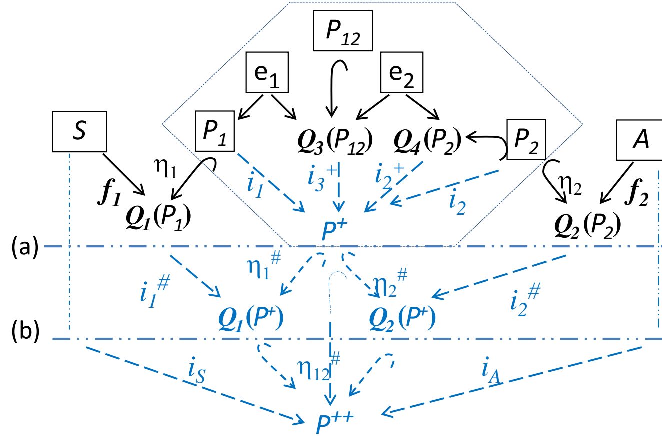


Fig. 7. The merge example, abstractly

The merged model is shown in Fig. 6; the merged metamodel is clear and implicit. Note the interplay between basic-derived-element links in mapping e_2. Without it, 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 our merge example in an abstract way. Nodes in the diagram denote models, and arrows are model mappings. Arrows with hooked tails denote inclusions of models into their augmentations with derived elements computed with queries Qi. Computed mappings are shown with dashed arrows, and computed model P^+ is not framed.

However, our system of models also has two view models, S and A, and to finish integration, we need to show how views S and A are mapped into the merge P. For this, we need to translate queries Q_1 and Q_2 to models P_1 and P_2 resp., 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 $\eta_k^\sharp$ within

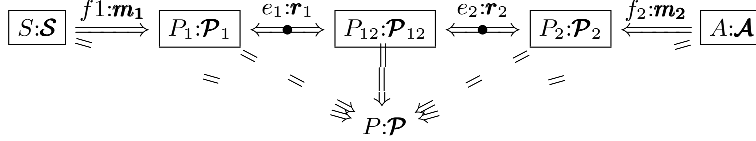


Fig. 8. Diagram from Fig. 7 in the Kleisli notation

the lane (a-b) in the figure. Finally, if query Q_k is monotonic, then as model P_k is embedded into model P^+, the result of executing Q_k against P_k can be embedded into the result of executing Q_k against P^+, hence we have mapping $i_k^\sharp$ making the square $[P_k\ P^+\ Q_k(P^+)\ Q_k(P_k)]$ commutative. Finally, we merge queries Q_1 and Q_2 into query Q_{12} (resulting in merge of the results produced by queries Q_k), and produce model P^{++} as shown below line (b) in the figure. We finish integration by building mappings $i_S\colon S \to P^{++}$ and $i_A\colon A \to P^{++}$ 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 model P^{++}.

2.3 The Kleisli construction

Diagram in Fig. 7 is precise but looks too detailed in comparison with informal diagram Fig. 1(b). Is it possible to design an abstract view of the detailed diagram combining precision and convenience? A positive answer is given by the Kleisli construction developed in category theory. The key idea is, given a mapping $f\colon X \to Q(Y)$, consider querying as a part of the mapping rather than the model Y, and write $(Q, f)\colon X \Rightarrow Y$ with Q being a query against Y and f a functional mapping $X \to Q(Y)$. We will call pairs (Q, f) as above *Kleisli mappings* and denote them by double-body arrows (to remind us that they encode both a query and a functional mapping). Then we do not need to designate different derived extensions of the same model, and diagram Fig. 7 becomes much more compact, as shown in Fig. 8. To make the diagram even more compact, we have applied one more standard categorical trick: spans e_1, e_2 are encoded by arrows with bullets (and can be indeed considered as morphisms in a special category).

Diagram Fig. 8 is a precise (and "almost formal") refinement of our original diagram Fig. 1(b): in contrast to the latter, the arrows in the former are Kleisli mappings and can be unraveled into the precise diagram Fig. 7. However, both these diagrams are not yet truly formal because we still did not formally define what queries and extensions of models with derived elements are. We will do this 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 [8]. 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 another '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' refers to some arrow $a\colon b \to$ Ann in 'graph' D_A, which is mapped by t_A to arrow *dom*: name $\to$ *string* in 'graph' M_A, but names of a and b are not essential for us. Details can be found in [9, 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 $\boldsymbol{p}$: ***Mod*** $\to$ ***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 (illustrated in Section 2). We will 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 in diagram Fig. 9(a). If an element $e \in N$ 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 $t{\restriction_v}\colon D{\restriction_v} \to N$ 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

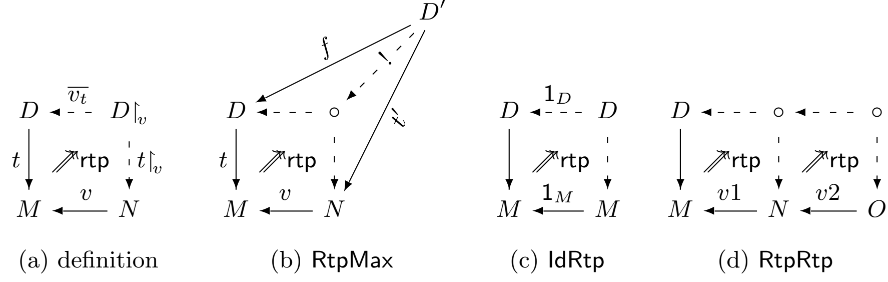


Fig. 9. Retyping operation: an application instance (a) and properties (laws) (b,c,d)

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 $\boldsymbol{p}\colon \boldsymbol{Mod} \to \boldsymbol{MMod}$. 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., [10, 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 $\boldsymbol{Mod}$ are called *models* and *model mappings*; elements of $\boldsymbol{MMod}$

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 Sect. 4.1 we reformulate constructions considered previously in precise yet still not entirely formal terms, and state several important laws for a reasonable query mechanism. Then we proceed with formalization based on monads (Sect. 4.2).

4.1 Queries via tile algebra

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 $Q_{\rm def}$ against metamodel M can be seen as an inclusion $\eta_M^{Q_{\rm def}}\colon M \hookrightarrow M.Q_{\rm def}$ of M into its augmentation with derived elements.

$$\begin{array}{ccc} D_A & \overset{\eta_A^{Q_{\rm exe}}}{\hookrightarrow} & D_A.Q_{\rm exe} \\ {\scriptstyle t_A}\downarrow & \mathsf{qExe} & \downarrow{\scriptstyle t_A.Q_{\rm exe}} \\ M & \underset{\eta^{Q_{\rm def}}}{\hookrightarrow} & M.Q_{\rm def} \end{array} \quad \text{or} \quad \begin{array}{ccc} \bullet & \hookrightarrow & \circ \\ {\scriptstyle A}\downarrow & \mathsf{qExe}[Q] & \downarrow \\ \bullet & \hookrightarrow & \bullet \end{array}$$

Fig. 10. Query execution: the arity shape

Execution of $Q_{\rm def}$ for an M-model A can be seen as an operation specified in Fig. 10 (left). Its input consists of two arrows, query definition $\eta^{Q_{\rm def}}$ 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.

$$\begin{array}{ccc} D & \overset{1_D}{\hookrightarrow} & D \\ {\scriptstyle t}\downarrow & \mathsf{qExe}[\mathsf{id}M] & \downarrow \\ M & \underset{1_M}{\hookrightarrow} & M \end{array} \qquad \begin{array}{ccccc} \bullet & \hookrightarrow & \circ & \hookrightarrow & \circ \\ \downarrow & \mathsf{qExe}[Q1] & \downarrow & \mathsf{qExe}[Q2] & \downarrow \\ \bullet & \hookrightarrow & \bullet & \hookrightarrow & \bullet \end{array}$$

(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 $\eta_{\mathsf{id}M}^{\rm def}$ 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 $\eta^{(Q1;Q2)_{\rm def}}\colon M \to M.Q1_{\rm def}.Q2_{\rm def}$, 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.

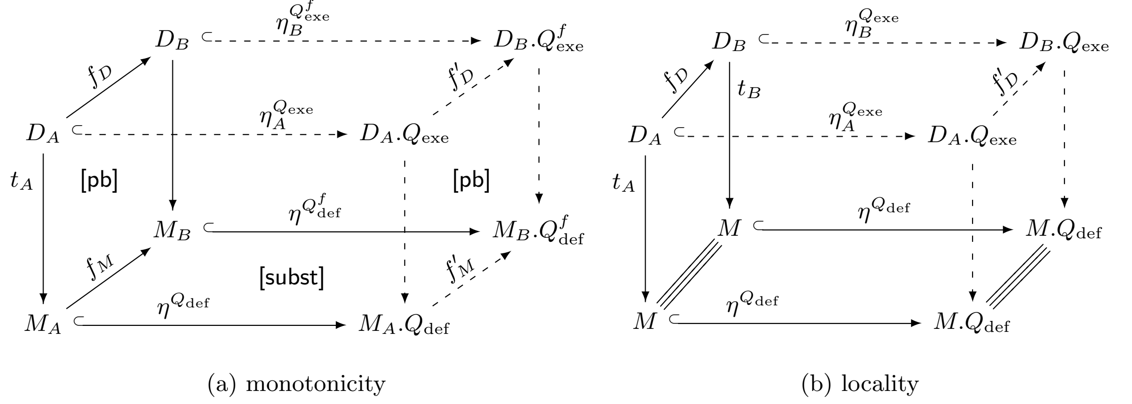
There are several other important properties of the query mechanism. Queries interact with model mappings, and if query Q is monotonic (i.e., preserves inclusion of datasets), then a model mapping $f\colon A \to B$ gives rise to a model mapping $f.Q\colon A.Q \to B.Q$. A detailed discussion of this and other properties can be found in the TR (Sect. 4.1, 4.2).

Query translation. Let Q be a query to model A as described by the front face of the cube in Fig. 12(a). 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 [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 $f'_M\colon M_A.Q_{\text{def}} \to M_B.Q_{\text{def}}$ and commutative bottom square of the cube. We may say that query Q_{def} is applied to metamodel mappings too, and write $f.Q_{\text{def}}$ for f'_M. Evidently, given two consecutive mappings $f\colon M \to N$, $g\colon N \to O$ between metamodels, we have $(f;g).Q_{\text{def}} = (f.Q_{\text{def}});(g.Q_{\text{def}})$. Also, $1_M.Q_{\text{def}} = 1_M$.

Interrelation between executions, $D_A.Q_{f\text{exe}}$ and $D_B.Q_{f\text{exe}}$, 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 $D_A.Q_{\text{exe}}$ is an inverse image of $D_B.Q^f_{\text{exe}}$ 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. 12(a), and the right face is a pullback too.

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 a subgraph of D_B as specified by the left face of cube Fig. 12(b). This square can only be 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. A query is said to be *monotonic*, if it preserves daataset inclusion, i.e., given injection $f_D\colon D_A \to D_B$, there is an injection $f_D.Q_{\text{exe}} = f'_D\colon D_A.Q_{\text{exe}} \to D_B.Q_{\text{exe}}$ making the entire cube commuta-



(a) monotonicity (b) locality

Fig. 12. Laws of query translation

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

Now we note that any heterogeneous model mapping $f = (f_M, f_D)\colon A \to B$ 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 Fig. 12(a,b) together provide existence of mapping $f.Q_{\rm exe}\colon D_A.Q_{\rm exe} \to D_B.Q_{\rm exe}$ for a local monotonic query Q. It is a standard categorical exercise in diagram chasing to see that for two consecutive model mappings $f\colon A \to B$, $g\colon B \to C$, we have $(f;g).Q_{\rm exe} = (f.Q_{\rm exe});(g.Q_{\rm exe})$ (assuming that query Q is monotonic and local). Also, $1_A.Q_{\rm exe} = 1_{A.Q_{\rm exe}}$.

4.2 Query mechanism via monads and fibrations

Query languages are monads. So far, given a query language QL over a metamodeling framework $\boldsymbol{p}\colon \boldsymbol{Mod} \to \boldsymbol{MMod}$, we have been considering individual QL-queries $\eta_A^Q\colon A \hookrightarrow A.Q$ (consisting of definition and execution as shown in Fig. 11(a)). It is technically convenient to merge all these into one huge "query" $\eta_A^{\mathsf{Q}}\colon A \hookrightarrow A.\mathsf{Q}$, 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 $\mathsf{Q}\colon \boldsymbol{Mod} \to \boldsymbol{Mod}$. Moreover, if mapping $f\colon A \to B$ is a pullback square, then mapping $f.\mathsf{Q}\colon A.\mathsf{Q} \to B.\mathsf{Q}$ is a pullback square as well (recall Fig. 12(b) and its discussion on Sect. 4.1).

Sequential query composition Fig. 11(c) gives rise to a mapping $\mu_A^{\mathsf{Q}}\colon A.\mathsf{Q}.\mathsf{Q} \to A.\mathsf{Q}$. 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 a commutative diagram in Fig. 13. Note also all elements in model $A.\mathsf{Q}_{\rm exe}.\mathsf{Q}_{\rm exe}$ are, in fact, already computed in $A.\mathsf{Q}_{\rm exe}$ and thus are copied from $A.\mathsf{Q}_{\rm exe}$ to $A.\mathsf{Q}_{\rm exe}.\mathsf{Q}_{\rm exe}$ and correspondingly re-

$$\begin{array}{ccc} D_A.\mathsf{Q}_{\rm exe} & \xleftarrow{\mu^{\mathsf{Q}_{\rm exe}}_{A.\mathsf{Q}}} & A.\mathsf{Q}_{\rm exe}.\mathsf{Q}_{\rm exe} \\ \downarrow & & \downarrow \\ M_A.\mathsf{Q}_{\rm def} & \xleftarrow{\mu^{\mathsf{Q}_{\rm def}}_{A.\mathsf{Q}}} & M_A.\mathsf{Q}_{\rm def}.\mathsf{Q}_{\rm def} \end{array}$$

Fig. 13.

typed according to mapping $\mu^{\rm def}_{A.\mathsf{Q}}$. That is, the diagram we discuss is nothing but 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 ***Mod***, and η^{Q}, μ^{Q} two mappings that assign to any model $A \in$ ***Mod*** model mappings η^{Q}_M and μ^{Q}_A as above. It is straightforward to check that we should also require commutativity of three (standard for categorical algebra) diagrams in Fig. 14.

$$\begin{array}{ccc} A.\mathsf{Q}^3 & \xrightarrow{\mu^{\mathsf{Q}}_{A.\mathsf{Q}}} & A.\mathsf{Q}^2 \\ {\scriptstyle \mu^{\mathsf{Q}}_A.\mathsf{Q}}\downarrow & & \downarrow{\scriptstyle \mu^{\mathsf{Q}}_A} \\ A.\mathsf{Q}^2 & \xrightarrow{\mu^{\mathsf{Q}}_A} & A.\mathsf{Q} \end{array} \qquad \begin{array}{ccccc} A.\mathsf{Q} & \xrightarrow{\eta^{\mathsf{Q}}_{A.\mathsf{Q}}} & A.\mathsf{Q}^2 & \xleftarrow{\eta^{\mathsf{Q}}_A.\mathsf{Q}} & A.\mathsf{Q} \\ & {\scriptstyle 1_{A.\mathsf{Q}}}\searrow & \downarrow{\scriptstyle \mu^{\mathsf{Q}}_A} & \swarrow{\scriptstyle 1_{A.\mathsf{Q}}} & \\ & & A.\mathsf{Q} & & \end{array}$$

(a) (b1) (b2)

Fig. 14. Laws for a query monad

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 ***Mod***. Importantly, constituting mappings of this monad have special properties related to pullbacks. To wit: for any model A, mappings $\eta^{\mathsf{Q}}_A \colon A \to A.\mathsf{Q}$ and $\mu^{\mathsf{Q}}_A \colon A.\mathsf{Q}.\mathsf{Q} \to A.\mathsf{Q}$ are pullback squares (the former is due to the requirement of data preservation, and the latter is just discussed). Recall also that functor Q preserves pullback squares. Since

Query execution vs. query definition. Our discussion in Sect. 4.1 and Fig. 11(a) show that a fundamental feature of querying (in contrast to updating) is that query execution always "run" over the respective query definition. We formalize this property by (a) postulating a query definition monad $(\mathsf{Q}_{\rm def}, \eta^{\mathsf{Q}_{\rm def}}, \mu^{\mathsf{Q}_{\rm def}})$ 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) $A.\mathsf{Q}.\boldsymbol{p} \ = A.\boldsymbol{p}.\mathsf{Q}_{\rm def}$, (b2) $\eta^{\mathsf{Q}}_A.\boldsymbol{p} \ = \eta^{\mathsf{Q}_{\rm def}}_A$, and (b3) $\mu^{\mathsf{Q}}_A.\boldsymbol{p} \ = \mu^{\mathsf{Q}_{\rm def}}_A$

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 $(\mathsf{Q}, \mathsf{Q}_{\rm def})$ over categories

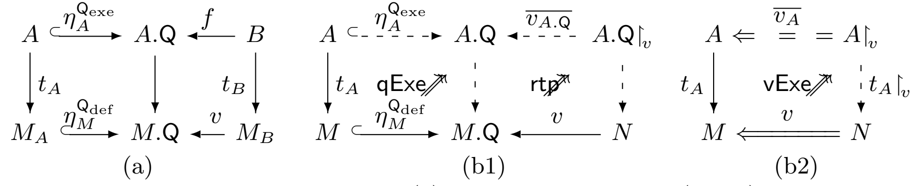


Fig. 16. Q-mappings (a) and view mechanism (b1,b2)

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.

4.3 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 $f\colon Y \Rightarrow X$ is a $\boldsymbol{C}$-arrow $f\colon Y \rightarrow X.\mathsf{Q}$. Composition of $\boldsymbol{C}_{\mathsf{Q}}$-arrows, say, $g\colon Z \rightarrow Y.\mathsf{Q}$ and $f\colon Y \rightarrow X.\mathsf{Q}$, is not immediate since f's source and g's target do not match. It is defined as shown in Fig. 15: $g;f\colon Z \Rightarrow X$ in $\boldsymbol{C}_{\mathsf{Q}}$ is $g;f^{\#}\colon Z \rightarrow X.\mathsf{Q}$ 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 $\eta^{\mathsf{Q}}_X\colon X \rightarrow X.\mathsf{Q}$. Monad axioms guarantee associativity of composition and unitality of identity, thus, $\boldsymbol{C}_{\mathsf{Q}}$ is indeed a category.

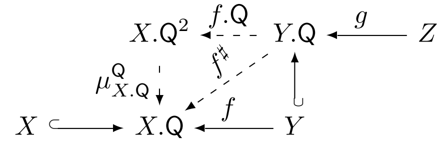


Fig. 15. The Kleisli construction

Lemma 1 ([11]). *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 $(\mathsf{Q}, \mathsf{Q}_{\rm def})$ be a monotonic query language over a metamodeling framework $\boldsymbol{p}$: ***Mod*** $\rightarrow$ ***MMod***. Arrows in the Kleisli category $\boldsymbol{Mod}_{\mathsf{Q}}$ are shown in Fig. 16(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. 8

Metamodel-level components of q-mappings between models are arrows in $\boldsymbol{MMod}_{\mathsf{Q}_{\rm def}}$, and they are nothing but view definitions: they map elements of the source metamodel to queries against the target one. Hence, we denote $\boldsymbol{MMod}_{\mathsf{Q}_{\rm def}}$ by $\boldsymbol{viewDef}_{\mathsf{Q}_{\rm def}}$ (and Lemma 1 is applicable again). View definitions can be executed as shown in Fig. 16(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. 16(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* $\boldsymbol{p}\colon \boldsymbol{Mod} \to \boldsymbol{MMod}$. *It gives rise to a split fibration* $\boldsymbol{p}_{\mathsf{Q}}\colon \boldsymbol{qMap}_{\mathsf{Q}} \to \boldsymbol{viewDef}_{\mathsf{Q}_{\mathrm{def}}}$ *between the corresponding Kleisli categories (and hence the view execution mechanism satisfies the laws in Fig. 17(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. 16(b2) (residing in $\boldsymbol{qMap}_{\mathsf{Q}}$) are $\boldsymbol{p}_{\mathsf{Q}}$-Cartesian, which defines the lifting operation. Law Fig. 17(a) is implied by the definition of identity loops in $\boldsymbol{qMap}_{\mathsf{Q}}$(which are inclusions η_A^{Q} in $\boldsymbol{Mod}$) and law Fig. 11(a). To prove Fig. 17(b), consider sequential Kleisli composition defined in Fig. 15. If f is Cartesian (prefix $\boldsymbol{p}$ is skipped), then $f.\mathsf{Q}$ is also Cartesian by Cartesianity of Q. Arrows μ_A^{Q} are always Cartesian, and hence arrow $f^\sharp = f.\mathsf{Q}; \mu_A^{\mathsf{Q}}$ 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 $\boldsymbol{qMap}_{\mathsf{Q}} = \boldsymbol{Mod}_{\mathsf{Q}}$. □

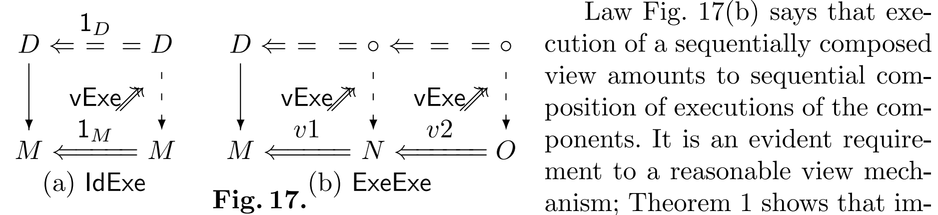


Fig. 17.

Law Fig. 17(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.

5 Related work

Modeling inductively generated syntactic structures (term and formula algebras) by monads and Kleisli categories is well known, e.g., [12, 13]. Semantic structures (algebras) then appear as Eilenberg-Moore algebras of the monad. In our approach, carriers of algebraic operations stay within the Kleisli category. It only works for monotonic query languages but the latter form a big practically interesting class. 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 the institution theory [14]. Indeed, inverting the projection functor gives us a functor $\boldsymbol{p}_{\mathsf{Q}}^{-1}\colon \boldsymbol{viewDef}_{\mathsf{Q}_{\mathrm{def}}}^{op} \to \mathbf{Cat}$, 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 easy and can be found in [15]. Conversely, there are attempts to add query facilities to institutions via so called *parchments* [16]. 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 [17] 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 a future work.

6 Conclusion

We showed that q-mappings, which may link elements in the source model to queries against the target model, are important for intermodeling and provide concise yet clear specifications for intermodeling scenarios. However, composition of such 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 showed that both syntax and semantics of a monotonic query language can be modeled by a Cartesian monad over the fibration of models over their metamodels. Then q-mappings can be composed within the Kleisli category of the monad.

In addition, viewing the universe of models and q-mappings $\boldsymbol{qMap}_{\mathsf{Q}}$ as a (Kleisli) category makes it a carrier of useful algebraic operations and design patterns developed in category theory. For example, in the paper we described model merging as colimit, but other operations and patterns may be applicable as well. An elaborated example can be found in our paper [17], where views are central for specifying consistency of heterogeneous multimodels.

The same argument works in one abstraction step “up”: if we formulate a metamodeling framework as a fibration $\boldsymbol{p}_{\mathsf{Q}}\colon \boldsymbol{qMap}_{\mathsf{Q}} \to \boldsymbol{viewDef}_{\mathsf{Q}_{\rm def}}$, we get a standard notion of mapping between frameworks for free. To wit, we define a framework mapping $m\colon \boldsymbol{p}_{\mathsf{Q}1} \to \boldsymbol{p}_{\mathsf{Q}2}$ as a fibration morphism, i.e., a pair of functors, $m'\colon \boldsymbol{viewDef}_{\mathsf{Q1}_{\rm def}} \to \boldsymbol{viewDef}_{\mathsf{Q2}_{\rm def}}$ and $m''\colon \boldsymbol{qMap}_{\mathsf{Q}1} \to \boldsymbol{qMap}_{\mathsf{Q}2}$, compatible with the monad and the Cartesian structure, and commuting with $\boldsymbol{p}_{\mathsf{Q}1}, \boldsymbol{p}_{\mathsf{Q}2}$. The universe of metamodeling frameworks is then converted into a category and a carrier of useful algebraic operations and patterns. We thus make heterogeneous intermodeling in-the-large also amenable to algebraic specification techniques and machinery.

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. Batory, D.S., Azanza, M., Saraiva, J.: The objects and arrows of computational design. In: MoDELS. Volume 5301 of LNCS., Springer (2008) 1–20
3. José Fiadeiro: Categories for Software Engineering. Springer (2004)
4. Sabetzadeh, M., Easterbrook, S.M.: View merging in the presence of incompleteness and inconsistency. Requir. Eng. **11**(3) (2006) 174–193
5. Romero, J., Jaen, J., Vallecillo, A.: Realizing correspondences in multi-viewpoint specifications. In: EDOC, IEEE Computer Society (2009) 163–172

6. Bernstein, P.: Applying model management to classical metadata problems. In: Proc. CIDR'2003. (2003) 209–220
7. 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 .
8. Ehrig, H., Ehrig, K., Prange, U., Taenzer, G.: Fundamentals of Algebraic Graph Transformation. (2006)
9. Diskin, Z.: Model synchronization: mappings, tile algebra, and categories. In R. Lämmel et al., ed.: Postproceedings GTTSE 2009. LNCS#6491, Springer (2011)
10. Jacobs, B.: Categorical logic and type theory. Elsevier Science Publishers (1999)
11. Manes, E.: Algebraic Theories. Springer (1976)
12. 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
13. Moggi, E.: Notions of computation and monads. Information and Computation **93**(1) (1991) 55–92
14. Goguen, J., Burstall, R.: Institutions: Abstract model theory for specification and programming. Journal of ACM **39**(1) (1992) 95–146
15. Diskin, Z.: Towards generic formal semantics for consistency of heterogeneous multimodels. Technical Report GSDLAB 2011-02-01, University of Waterloo (2011)
16. 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
17. 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