SKETCHES, QUERIES, VIEWS, AND KLEISLI COMPOSITION:
TOWARDS UNIVERSAL ALGEBRA OF DIAGRAMMATIC
OPERATIONS WITH PRE- AND POST-CONDITIONS.

ZINOVY DISKIN AND PATRICK STÜNKE

McMaster University, Centre for Software Certification, Hamilton, Canada
e-mail address: diskinz@mcmaster.ca

Høgskulen på Vestlandet, Institutt for Datateknologi, Elektroteknologi og Realfag, Bergen, Norway
e-mail address: past@hvl.no

Abstract. A diagrammatic operation takes a diagram of a specified shape as its input
and completes it with new elements forming a bigger diagram. Pre- and post-conditions
are an important part of the story (e.g., monic preservation by pullbacks) so that diagram
chasing smoothly integrates logic and algebra. This technique transcends category theory
and finds numerous applications in practice of software engineering, specifically in the
multiview approach to system design, and in model management (in the sense of model-
driven engineering). These applications force us to consider a very general version of
diagram chasing over generalized sketches (in the sense of Makkai) and give rise to a project
reported in the paper (an accompanying TR can be found online).

1. Introduction

A diagrammatic operation takes a diagram of a specified shape as its input, and extends
it with new elements forming a bigger diagram of a specified shape; we will say that the
input diagram is completed by the operation. For example, arrow composition takes two
consecutive arrows and completes them with a third arrow forming the corresponding triangle.
The resulting diagram often satisfies some postconditions, e.g., a chosen pullback completes
a cospan with a commutative square, and a chosen powerobject completes an object with its
jointly-monic membership span. Preconditions are needed too, e.g., adhesive categories only
provide pushouts for one-leg-monic spans (and then we know that the corresponding leg of
the resulting cospan is also monic). Thus, categorical diagram operations come together with
reasoning over them so that diagram algebra and logic are smoothly integrated in diagram
chasing routines.

This paper stems from the observation that the diagram chasing idea transcends Category
Theory (CT) and finds numerous applications in the practice of Software Engineering (SE),
in which, however, it is currently used in an ad-hoc way: implicit, semi-formal or formal
but buried in code so that the underlying diagram chasing procedures are blurred and
distorted. The negative impact of these issues increases with the recent advent of Model

Key words and phrases: Software Modeling, Presheaves, Sketches, Diagrammatic Operations, Diagrammatic Reasoning, Kleisli Composition.
Driven Engineering (MDE), in which diagrammatic models rather than code become the primary artifacts of software development. Our goal in the project reported in the present paper\(^1\) is to provide an accurate formalization of diagrammatic reasoning routines adequate to their practical use in MDE. We will discuss this in more detail in the next section along with several mathematically interesting structures and results emerged along the course, and our contributions. Sections 3 and 4 provide basic formal definitions and results, and Sect. 5 concludes. Appendix A is optional but helpful: it illustrates main motivating ideas with a simple example.

2. Overview

We discuss two main applications of the framework in SE in Sect. 2.1-2, then discuss some mathematical aspects of the framework in Sect. 2.3, and describe related work and contributions in Sect. 2.4

2.1. Views over sketches. Developing a complex software system normally requires working with multiple views [ASB10] of the system: This is the only way to manage complexity and a cornerstone of system engineering, see [FKN\(^+\)92] for an early discussion and [FKWVH19, BBCW19, CCP19] for more recent reports and surveys. A view of a source artifact \(S\) is, in general, not merely a part of \(S\) or a projection of \(S\), but typically involves some computation over \(S\), e.g. composing references, aggregating attributes by summing up their values etc, which extends \(S\) with derived elements produced by some predefined operations (we will say queries) applied to \(S\). Then a view can be defined as a mapping \(v: V \rightarrow Q_v(S)\) in a suitable category of software artifacts where the source \(S\) and the view schema \(V\) live, and \(Q_v\) is a query that provides an extension \(Q_v(S)\) of source \(S\) with derived elements required by the view. Note an important distinction from the ordinary Kleisli mappings defined for a given monad \(Q\) or, at least, a global object mapping, which only potentially exists and seems unnecessary abstract for software engineers not familiar with Haskell. In contrast, in our notion of view, each view mapping \(v\) carries its own query \(Q_v\) and maps to its own local extension \(Q_v(S)\) rather than to a unified but huge global extension \(Q(S)\). A simple example in Appendix A shows how it works, and also demonstrates the importance of constraints for the view mechanisms. Thus, the basic category hosting models like \(S\) and \(V\) is a category of generalized sketches over graphical structures rather than pure graphs.

2.2. Algebra for model management. In model management, models are manipulated as holistic entities, and replacement the one-object-at-a-time approach (micromanagement) by one-model-at-a-time (macromanagement) turned out very productive [Ber03, FN05, PKR\(^+\)13]. It requires considering high-level operations on models such as model slicing, transformation, merging, and synchronization, and composing them into complex workflows [DKM13].

\(^1\) which is an extended abstract of ongoing work in progress, whose current state is reported in the technical report [DS20](further referred to as “the TR”)
Fig. 1 shows a simple example in which two views of the same source are first matched, and then merged. One may think of this as taking the pullback and then the pushout, but in practice these “pure” categorical operations are often interwoven with some normalization procedures (and may require conflict resolution) so that even in this simple case the real workflow will be much richer. Importantly, models in SE are typically considered up to their auto-isomorphisms generated by permutations of OIDs (in the sense of OO programming) which are irrelevant and normally not visible. It justifies making categories of models skeletal and having only identical automorphisms, which makes limits and colimits deterministic algebraic operations.

Thus, for both the inter-model and the intra-model levels of reasoning, SE needs powerful and formally specified algebraic machinery for diagrammatic operations. On the other hand, it seems that even basic universal algebraic constructs (signature, term, parsing) have not been accurately specified for diagrammatic operations, especially when the latter are considered together with pre-and post-conditions and hence are interwoven with logic; in this sense, we are developing some basic universal algebra for diagrammatic operations.

2.3. Mathematical aspects.

2.3.1. Why generalized graphs and generalized sketches. As discussed above, categories hosting software models are categories of sketches. In general, a sketch is a graphical structure endowed with labelled diagrams (a.k.a. constraints); in the project, we consider a very general version of the notion: our ‘graphs’ are finitary presheaves, and our constraints of shape $G$ are any “devices” that classify mappings from $G$ as either valid or invalid. The generality of our graphs is caused by the great diversity and complexity of graphical structures used in SE [Ken02], and graphs used for behavioural and workflow models can even be more complex. Similarly, constraints used in domain modelling (banking, e-commerce, automotive etc.) can be complex and very element-wise-oriented so that their representation in a standard categorical way via universal properties can be rather bulky and unwieldy (a detailed discussion can be found in [DW07], see also Appendix). Thus, we need constraints and sketches of a very general nature that Michael Makkai called generalized sketches [Mak97].

2.3.2. String diagrams do not work for diagram operations. For PROPs, terms can be specified by string diagrams as shown in Fig. 2, which depicts a ternary term $(x + y) \ast (x + z)$ built from two binary operations. Note that reuse of the middle variable $y$ is shown explicitly with Copy-operation (visualized via $\rhd$). This is possible as set $\{x, y, z\}$ is the coproduct of
its elements \( \{x\} \sqcup \{y\} \sqcup \{z\} \). However, diagram operations are graph-based, and a graph (while being a colimit of its elements) is not their coproduct!

Consider Fig. 3 presenting a simple diagram chasing piece. Part (A) uses a typical categorical notation: the diagram presents a workflow consisting of two pullbacks taken in parallel over spans with one monic leg, followed by yet another pullback. Monic preservation by pullbacks is also shown. Part (B) makes the above explicit by using labels for constraint declarations (ovals) and operation applications (chevrons), and showing their scope. Note that arrows going in and out of chevrons show the workflow and what elements are constructed/derived (blue and dashed) rather than given. Part (C) is similar to Fig. 2 and highlights the workflow of the query by making the order of operation applications explicit, e.g. the pullback \( c \) depends on elements produced by the other two pullbacks.

Note several important differences with Fig. 2. First, in discrete PROP-setting, the elements produced by an operation have no further relationship with the input elements, while in the graph-based setting, there is an overlap between the input and the output, e.g. element \( B \) in Fig. 3(B) is both in the input and the output of the pullback application \( a \), and similarly for application \( b \) so that \( B \) is also shared by \( a \)-output and \( b \)-output. Second, the workflow shown in Fig. 3(C) is a rather poor view of the actual workflow under the hood, which consists of several pushouts (POs): three POs are needed to produce graph \( X' \) and yet another one to produce \( X'' \). Workflow arrows with hollow ends are, in fact, pairs, \( (X,a) \), \( (X,b) \) etc., while mappings that do the job, e.g., the one showing what part of graph \( X \) is used as an input for \( a:[pb] \), are not shown. Thus, in contrast to Fig. 2, diagram Fig. 3(C) is an incomplete specification of the story.
2.4. Related Work and Contributions. By logic we refer to work on generalized sketches, and algebra refers to work on signatures of operation symbols, terms and algebras over them.

2.4.1. Logic. Several authors noticed that Ehresmann's sketches can be applied to data modelling [LS91, PS95, JR02]. However, these sketches are based on universal properties of the corresponding (limit, colimit, powerobject) diagrams, which (as we discussed above) is a too restrictive setting for applications. A suitable generalization of sketches by exactly this reason was independently introduced by Diskin [Dis96, Dis97], and even earlier, in a very different context of an abstract approach to logic, by Makkai [Mak97] (whose preprints had been circulating in early 90s). Makkai developed a coherent theory of generalized sketches and proved a fundamental result that any category of sketches is a presheaf topos. Hence, sketches in a signature $\mathcal{P}_1$ can be considered graphs over which a new signature $\mathcal{P}_2$ and hence a new category of $\mathcal{P}_2$-sketches can be defined, and so on. In this way, Makkai specified categories with several layers of additional structure as sketches. In our project, we use a different approach in which all discrete signatures $\mathcal{P}_i$ are integrated into one signature-category $\bigcup_i \mathcal{P}_i$, whose arrows describe dependencies between predicate symbols extracted from arities (details can be found in [DS20, Sect.3]). In the TR we prove that Makkai's theorem can be extended for categorical signatures so that the slogan “sketches are graph” persists (see Th. 3.7 below for a precise formulation). Also, while traditionally sketches are defined in an indexed way as families of sets of diagrams indexed by predicate symbols (for the discrete signature case), or functors from the signature (for the categorical signature case [DW07]), we define them in an equivalent fibrational way via a discrete opfibration functor into the signature: the fibrational definition is more compact and often simplifies technicalities.

2.4.2. Algebra. Diagram operation signatures and algebras were defined first in [Dis97] and with some additional details in [Dis96], but no mathematical results were obtained. For the special case of operations over ordinary graphs, Wolter et al [WDK18] proved that any signature of such operations generates a monad (i.e., a global construct) over the category of graphs. The machinery of local Kleisli mappings explained informally was used in several papers of the first author, e.g., [DXC10, DGC17], and formalized as Kleisli mappings of a monad (i.e., globally) in [DMC12]; we are not aware of an accurate formalization of Kleisli mappings considered locally as is done in our project, nor of several other results reported in Sect.4. Specifically, we prove that a) sketches and local Kleisli mappings (views) between them form a symmetric monoidal category, and b) the local Kleisli composition gives rise to a Kleisli triple and hence a monad (as expected).

3. Diagram Logic: Sketches

3.1. Signatures with dependencies and sketches over them. As our discussion in Sect.2 and Example in Appendix suggest, we need a very general notion of graphs as presheaves $G: \mathcal{G} \to \textbf{Set}$ for a suitable schema category $\mathcal{G}$. In all our applications, graphs have hierarchical structure that allows their traversal from the top elements to the bottom. For example, ordinary directed multigraphs are given by schema $\mathcal{G} = \text{Graph}$ with $\text{Ob(\text{Graph})} = \{\text{Node}, \text{Arrow}\}$ and two non-identity arrows $\text{src}, \text{trg}: \text{Arrow} \to \text{Node}$ so that object $\text{Arrow}$ is above $\text{Node}$. Other examples can be found in the TR. Hence, we require category $\mathcal{G}$ to be hierarchical in the following sense.
Definition 3.1 (Hierarchical categories). A *hierarchical category* (h-category in brief) is given by a surjective on objects functor hei: \( H \rightarrow [0, N]^{\text{op}} \), where \( H \) is a small category, and \( N \) is a natural number called the *height of \( H \)* with \([0, N]\) being the obvious linearly ordered set considered as a posetal category. Moreover, the following *acyclicity* condition holds for any two objects \( x, y \) in \( H \): if \( \text{hei}(x, y) \neq \emptyset \), then either \( \text{hei}(x) > \text{hei}(y) \) or \( x = y \), but in the latter case we require \( \text{hei}(x, x) = \{\text{id}_x\} \). Thus, all *layers* \( L_n(H) = \text{hei}^{-1}(n) \) for \( 0 \leq n < N \) are not empty, and all \( H \)-arrows go down from higher to lower layers while the only arrows within a layer are identities. The empty category is an h-category of height 0.

Definition 3.2 (Graphs). An \( \mathcal{G} \)-graph or just a graph is a functor \( G: \mathcal{G} \rightarrow \text{Set} \) whose domain \( \mathcal{G} \) (called the *schema*) is an h-category. For a graph \( G: \mathcal{G} \rightarrow \text{Set} \), elements of set \( G(H) \) with \( H \in \mathcal{G}_n \) are called n-cells (of type \( H \)), and if \( \mathcal{G}_n \) is not a singleton, we say that n-cells are multi-sorted. We write \( \text{Elem}(G) \) for the set \( \bigcup_{H \in \text{Ob}(\mathcal{G})} G(H) \).

The presheaf topos of all \( \mathcal{G} \)-graphs will be denoted by \( \text{G} \), and we assume that throughout the paper the schema \( \mathcal{G} \) is arbitrary but fixed. A graph \( G \) is *finite* if \( \text{Elem}(G) \) is finite.

Definition 3.3 (Predicate Signature). Let \( \mathcal{G} \) be a schema category for graphs and \( \mathcal{G} = \text{Set}^\mathcal{G} \) is the corresponding presheaf topos. A *predicate signature* \( \mathcal{P} \) over \( \mathcal{G} \) is given by a h-category \( |\mathcal{P}| \), whose objects are called *predicate symbols* and arrows are *dependencies*, and an *arity* functor \( \alpha: |\mathcal{P}| \rightarrow \mathbb{C}^{\text{op}} \). Thus, if a symbol \( P \) depends on symbol \( P' \) via dependency \( p: P \rightarrow P' \), then there is a graph morphism \( p^\alpha: P^\alpha \leftarrow P'^\alpha \), where we write \( X^\alpha \) for \( \alpha(X) \) for an \( X \in \text{Ob}(|\mathcal{P}|) \) but we will use the bracketed notation too. For a predicate symbol \( P \in |\mathcal{P}| \), we say that graph \( P^\alpha \) is the *arity of \( P \)*. As a rule, we will (inaccurately) use symbol \( \mathcal{P} \) for both a signature and its carrier category \( |\mathcal{P}| \).

A signature is called *finitary* if schema \( \mathcal{G} \) is locally finite and all arity graphs \( P^\alpha, P \in \mathcal{P} \) are finite. This is what we have in applications, but our results below do not depend on (in)finitariness.

![Diagram](image)

(A) Category \( |\mathcal{P}_{\text{pbm}}| \)  (B) Arity \( [\text{monic}]^\alpha \)  (C) Arity \( [\text{pbm}]^\alpha \)  (D) Arity \( [\text{sqr}]^\alpha \)

**Figure 4.** A sample signature \( \mathcal{P}_{\text{pbm}} \)

Example 3.4 (Pullbacks over monics). Consider ‘graphs’ being ordinary graphs, and a predicate signature with three symbols, \( \text{Ob}(|\mathcal{P}|) = \{[\text{monic}], [\text{pbm}], [\text{sqr}]\} \), whose arities are as expected and shown in Fig. 4(b,c,d) resp. We assume these arities to be concrete fixed graphs whose elements are given fixed names formed by numeric strings (but we could also used colours or whatever names we like). We assume that predicate \( [\text{pbm}] \) can only be declared for squares in which arrows 10 and 32 are declared to be \([\text{monic}]\) as e.g., in adhesive categories. Later we will describe \([\text{pbm}]\) as an operation for which the requirement for arrow 10 to be monic will be a precondition while a similar requirement for arrow 32 will be a postcondition, but when \([\text{pbm}]\) is a predicate, both requirements are “postconditions”. Another
postcondition is the requirement for the entire square to be \([=_{\text{sqr}}]\) (read “commutative”). Then we partition \(\mathcal{P}\) into two layers, \(|\mathcal{P}|_0 = \{[\text{monic}], [=_{\text{sqr}}]\}\) and \(|\mathcal{P}|_1 = \{[\text{pb}_m]\}\) connected by dependency arrows \(p_1, p_2, p\) as shown in diagram (a). This gives us an \(h\)-category \(\mathcal{P}\) and a functor \(\alpha : \mathcal{P} \to \mathcal{G}_{\mathcal{G}}^{\text{op}}\). Note that mapping \(p^\alpha : [=_{\text{sqr}}]^\alpha \to [\text{pb}_m]^\alpha\) is an isomorphism rather than identity.

We need predicate symbols to declare atomic constraints, e.g., \(P(x, y)\) in ordinary first-order logic, and simplest logical theories are built from conjunctions of atomic formulas, e.g., \(P(x, y) \land P(y, z)\). A diagrammatic counterpart of conjunctive theories is the notion of a \textit{sketch}; a set of atomic formulas over a graph of variables, which is closed w.r.t. all dependency arrows in the signature.

**Definition 3.5** (Sketch over a signature \(\mathcal{P}\) with dependencies). A (generalized) sketch \(S\) over \(\mathcal{P}\) is given by a graph \(|S| \in \mathcal{G}\) called the carrier, and a span of two functors \(\text{label} \) and \(\text{diagr}\) making a commutative outer square in diagram (3.1). The apex of the span is a small category \(\text{Index}(S)\) of (diagram) indexes and dependencies; moreover, functor \(\text{label} : \text{Index}(S) \to |\mathcal{P}|\) is required to be a discrete opfibration so that all arrows in \(\text{Index}(S)\) are lifts of \(\mathcal{P}\)-arrows—this is the closure condition mentioned above (it is discussed in detail in [DW07]).

![Diagram](3.1)

We call the definition above \textit{fibrational} to distinguish it from a typical \textit{indexed} sketch definition, in which the discrete opfibration \(\text{label}\) is replaced by presheaf \([\_] : |\mathcal{P}| \to \text{Set}\) and \(\text{diagr}\) is a functor \([\_] : \mathcal{G}_{\mathcal{G}}^{\text{op}}/|S|\) (this is how sketches were defined in [DW07]).

A sketch is \textit{finitary} if the signature is finitary. A sketch is \textit{finite} if its carrier is a finite graph and category \(\text{Index}(S)\) is finite.

**Definition 3.6** (Sketch morphisms). Let \(S = (|S|, \text{label}, \text{diagr})\) and \(S' = (|S'|, \text{label}', \text{diagr}')\) be two sketches. A \textit{sketch morphism} \(f: S \to S'\) is a pair consisting of an \(\mathcal{G}\)-arrow \(|f| : |S| \to |S'|\) and a functor \(f_{\text{constr}} : \text{Index}(S) \to \text{Index}(S')\) such that \(f_{\text{constr}} \circ \text{label} = \text{label}\) and \(f_{\text{find}} \circ \text{diagr} = \text{diagr} \circ (\mathcal{G}/|f|)\) with \(\mathcal{G}/|f| : \mathcal{G}/|M| \to \mathcal{G}/|M'|\) defined by postcomposition with \(|f|\). This gives us the category of \(\mathcal{P}\)-sketches denoted by \(\mathcal{G}/|\mathcal{P}|\) (this is Makkai’s notation that turned out convenient).

A fundamental and not obvious property of the sketch construct (first noticed by Makkai for the case of a discrete signature without dependencies) is that sketches are, in fact, presheaves. To prove it for a signature with dependencies, we first note that currying functor \(\alpha : |\mathcal{P}| \to \text{Set}^\mathcal{G}\) gives us a profunctor, whose collage gives us a hierarchical category \(\mathcal{G} \star_{\alpha} \mathcal{P}\).

**Theorem 3.7** (Extended Makkai theorem). The category of \(\mathcal{P}\)-sketches \(\mathcal{G}/|\mathcal{P}|\) is equivalent to the presheaf topos \(\text{Set}^{\mathcal{G} \star_{\alpha} \mathcal{P}}\). In other words, \(\mathcal{P}\)-sketches over \(\mathcal{G}\)-graphs can be interpreted as \((\mathcal{G} \star_{\alpha} \mathcal{P})\)-graphs (and thus diagrammatic constraints are just higher-order cells).

**Remark 3.8.** The proof doesn’t actually use the fact that categories \(\mathcal{G}\) and \(\mathcal{P}\) are hierarchical and we thus can formulate the theorem for any small \(\mathcal{G}\) and \(\mathcal{P}\) (as Makkai did). However, in
our further work with operations, we will need the hierarchical structure of \( \mathcal{G} \) and \( \mathcal{P} \); note also that \( \mathcal{P} \) is to be above \( \mathcal{G} \) if even they are not hierarchical themselves.

4. Basic Diagram Algebra

Below we assume given some fixed category of sketches \( \mathbb{S} = \mathbb{G} \upharpoonright \mathbb{P} \) (which, according to Makkai’s extended theorem, can be considered graphs).

**Definition 4.1** (Diagram Operation). Let \( A \in \mathbb{S} \) be a \( \mathcal{P} \)-sketch and \( m: I_m \rightarrow E_m \) a sketch monic. An operation over \( A \) of arity \( m \) is a (labelled) function \( \Phi_m: \mathbb{S}(I_m, A) \rightarrow \mathbb{S}(E_m, A) \) such that for any arrow \( b: I_m \rightarrow A \) (called binding), the triangle \((m, b, \Phi(b))\) commutes: \( m \circ \Phi(b) = b \). Commutativity condition is fundamental: it ensures that an operation does not change its input data.

Given another sketch \( A' \) with operation \( \Phi'_m \) of arity \( m \), a sketch morphism \( f: A \rightarrow A' \) is called \( m \)-compatible iff. \( b \circ f = m \circ \Phi(b \circ f) \) holds for every \( b: I_m \rightarrow A \).

**Definition 4.2** (Discrete Operation Signatures and Algebras). A discrete operation signature \( \mathcal{O} \) over \( \mathbb{S} \) is given by a set \(|\mathcal{O}|\) of operation symbols and an arity function \( \alpha: |\mathcal{O}| \rightarrow [\cdots, \mathbb{S}] \). For a symbol \( \phi \in |\mathcal{O}| \), we say that monic \( \phi^\alpha: I_\phi \rightarrow E_\phi \) is the arity of \( \phi \). We will also denote this monic by \( \phi^\alpha \) or \( IE_\phi \) and simply write \( \mathcal{O} \) for \(|\mathcal{O}|\).

A signature \( \mathcal{O} \) is finitary if \( \mathcal{P} \) is finitary and sketches \( I_\phi, E_\phi \) are finite for all \( \phi \in \mathcal{O} \). This is what we have in applications, but our results below do not depend on the signature’s (in)finitariness.

An algebra \( A \) for \( \mathcal{O} \) is given by a carrier sketch \(|A|\) and an operation \( [\phi]^A; \mathbb{S}(I_\phi, |A|) \rightarrow \mathbb{S}(E_\phi, |A|) \) for each \( \phi \in \mathcal{O} \) (which means commutativity of the corresponding triangle for each \( \phi \)). An algebra homomorphism is a sketch morphism compatible with all operations in \( \mathcal{O} \).

**Proposition 4.3** (Category of \( \mathcal{O} \)-algebras). There is a category \( \mathbb{S}^\mathcal{O} \) whose objects are algebras with \( \mathbb{S} \)-carriers and arrows are \( \mathcal{O} \)-homomorphisms.

**Example 4.4** (Universal algebra). Consider \( \mathcal{G} = 1 \) so that \( \mathbb{G} \cong \mathbb{Set} \), and \( \mathcal{P} = \emptyset \) so that \( \mathbb{G} \upharpoonright \mathbb{P} \cong \mathbb{G} \cong \mathbb{Set} \). Every “classical” operation signature \( \Omega \) with an arity function \( \text{ar}: |\Omega| \rightarrow \text{Nat} \) can be interpreted as a diagrammatic signature \( \mathcal{O}_\Omega = (|\Omega|, \alpha) \) such that for each symbol \( \omega \), its arity is given by the input set \( I_\omega = \text{ar}(\omega) \) considered as a set, the scope set \( E_\omega = I_\omega + \{*\} \) and the obvious monic \( m_\omega: I_\omega \rightarrow E_\omega \). Any \( \mathcal{O}_\Omega \)-algebra \( A \) is an \( \Omega \)-algebra \( \hat{A} \): let \( n = \text{ar}(\omega) \) and \( a \in A^n \), then setting \( [\omega]^\hat{A}(a) \overset{\text{def}}{=} [\omega]^A(a)(* \) defines an \( \Omega \)-algebra structure on \(|A|\). The converse construction from a classical \( \Omega \) algebra \( A \) to a diagrammatic \( \mathcal{O}_\Omega \) algebra \( \hat{A} \) is equally obvious: for any \( a \in A^n \) and any \( x \in I_\omega \subset E_\omega \), \( [\omega]^\hat{A}(x) \overset{\text{def}}{=} a(x) \), and \( [\omega]^\hat{A}(\ast) \overset{\text{def}}{=} [\omega]^A(\ast \). Moreover, \( \hat{A} \cong A \) and \( \hat{A} \cong A \). It is also easy to check that the categories \( \mathbb{S}^\mathcal{O}_\Omega \) and \( \Omega-\text{Alg} \) are isomorphic. Thus, the ordinary universal algebra is a specialization of the discrete diagrammatic algebra.

**Definition 4.5** (Atomic Queries (= atomic terms)).

Let \( \mathcal{O} \) be a discrete operation signature over a category of sketches \( \mathbb{S} = \mathbb{G} \upharpoonright \mathbb{P} \). An atomic query is a triple \( q = (I_q, \phi, b_q) \) of an input sketch \( I_q \in \mathbb{S} \), an operation symbol \( \phi \in \mathcal{O} \) (we omit the subscript \( q \)) and a binding sketch morphism \( b_q: I_\phi \rightarrow I_q \). These data determine pushout squares (4.1) in \( \mathbb{S} \), in which the resulting objects \( E_q \) is determined up to iso.

\[
\begin{array}{ccc}
I_\phi & \xrightarrow{IE_\phi} & E_\phi \\
\downarrow I_q & \quad & \quad \downarrow I_q \\
E_q & \xrightarrow{b_q} & E_q \\
\end{array}
\]
Remark 4.6 (Concrete Syntax). By fixing a concrete syntax for specifying terms, we fix a particular object $E_q$, while the general pushout formulation gives us a concrete-syntax-free description. Consider, for example, a binary operation $+$ in the ordinary universal algebra setting (Example 4.4) and an atomic query $q = (X, +, b_q)$ over a set of variables $X = \{x, y, z\}$ in the role of $I_q$ and a binding mapping $b_q: 2 \to X$, where $2 = \{0, 1\}$ and $b_q = \{0 \mapsto x, 1 \mapsto y\}$. The diagrammatic arity of $+$ is an inclusion $IE_+: \{0, 1\} \hookrightarrow \{0, 1, *\}$ with * being an arbitrarily chosen token, and taking pushout of $(IE_+, b_q)$ gives us a four-element set, e.g., $E_q = \{x, y, @, z\}$ with @ being a new element representing the application of $+$ to $(x, y)$ (indeed, $b^E_+(*) = @$ irrespective of what tokens are chosen for * and @). If elements of set $X$ are chosen to be strings (a typical case), then for the new element we may take a string $x + y$ (infix) or $+ (x, y)$ (prefix) or $xy+$ (reversed polish notation) etc. depending on the respective choice of the concrete syntax. Thus, choosing a concrete syntax for an atomic term $q$ amounts to choosing a concrete pushout square for the set of binding squares determined by $q$. However, we will not use any special properties of a chosen pushout square besides its universality, and hence our considerations are actually concrete-syntax-free, i.e., do not depend on the chosen concrete syntax (analogously to defining a function with an unspecified parameter).

Definition 4.7 (Multiqueries). A (finite) multiquery $Q$ over a discrete one-layer signature $\mathcal{O}$ (over a category of sketches $\mathcal{S} = \mathbb{G} || \mathcal{P}$) is given by a sketch $I_Q \in \mathcal{S}$ called the input, and a finite family of atomic queries with the same input sketch $I_Q$. This family is given by a span of two functions $\text{label}$ and $\text{diagr}$ making a commutative diagram (4.2b)

![Diagram](https://example.com/diagram.png)

(4.2)

We may now re-define the notion of algebra: An algebra $A$ over $\mathcal{O}$ is given by a carrier sketch $|A|$ and a mapping $\lfloor \_ \rfloor^A$, shown in diagram (4.2b), where the span $(\alpha^A, \gamma^E)$ is derived as a pullback in $\text{Set}$. Elements in its apex $\text{LDiagr}(A)$ can be identified with pairs $q = (\phi, \delta)$ in which $\delta$ is a pushout diagram in $\mathcal{S}$ with the left-bottom corner being $|A|$ and the upper monic being $\phi^A$ due to commutativity.

Function $\lfloor \_ \rfloor^A$ maps any such pair to a triangle diagram in $\mathcal{S}$, in which

\[2\text{Note that this diagram is in } \text{Set}.\]
• the left-side (binding) arrow is the same as in diagram $\delta$ due to the commutativity of the trapezoid $(a)$,
• the upper monic is $\phi^\alpha$ due to commutativity $\alpha^\beta_{|IE} = [\ldots]^A_{|IE}$;
• hence, the right-side arrow can be considered as the result of the operation: in fact, the right-side “formal” arrow in the pushout square $\delta$ freely added to graph $|A|$ is replaced by an actual arrow in $|A|$, namely, $[q]^A_{|IE}$.

Thus, diagrams at the target of arrow $[\ldots]^A$ are actually commutative triangles from Def. 4.1.

**Definition 4.8** (Complex queries and views). A complex query $Q$ is a finite sequences of multiqueries $[Q_1, \ldots, Q_n]$ such that $I_{i+1} = E_{Q_i}$. Hence, we have a sequence of monics

$$I_Q = I_{Q_1} \Rightarrow_{IE_{Q-1}} \ldots \Rightarrow_{IE_{Q_n}} E_{Q_n} = E_Q$$

whose composition will be denoted by $IE_Q$. This arrow will alternatively be denoted as $\eta_Q$ and we call it the unit for $Q$. The number $n$ is the length of the query.

Note that a multiquery $Q'$ can be interpreted as a complex query of length 1.

Let $S$ be a $P$-sketch called the source, and $O$ an operation signature over $P$. An $O$-view of $S$ is a pair $v = (Q_v, \hat{v})$ with $Q_v$ a complex $O$-query over $S$ and $\hat{v}: Q(S) \leftarrow V$ a sketch morphism called view (definition) mapping; the domain of the mapping is called the view schema. Thus, a view is a cospan $S\xrightarrow{\eta_Q} Q_v(S) \leftarrow V$, which we will write as

$S\xrightarrow{\eta_v} Q_v(S) \xleftarrow{\hat{v}} V$ to ease notation. Note that the left leg is monic while the right one is arbitrary. We will also write views as $v: V \rightarrow S$ or $v_Q: V \rightarrow S$ to make the query explicit.

**Theorem 4.9** (Category of Views (local Kleisli construction)). Views are associatively composable and thus make a category $\mathbb{S}_O$ whose objects are sketches in $\mathbb{S} = \mathbb{G}||P$, and arrows are $O$-views between them.

**Theorem 4.10** (Views form a SMC). The category of $O$-views $\mathbb{S}_O$ has a symmetric monoidal structure $(\otimes, I)$ (via coproducts and initial objects in $\mathbb{S}$).

Given a sketch $A \in \mathbb{S}$, let $QQ(A)$ be the set of all complex queries $Q$ over $A$ with $I_Q = A$, and $Q(A)$ is the colimit of the corresponding (infinitely) wide span $(\eta_Q)_{Q \in QQ(A)}$ yielding an obvious sketch morphism $\eta_A: A \rightarrow Q(A)$. Along these lines, we prove the following.

**Theorem 4.11** (Query Monad). Any operation signature $O$ over $\mathbb{S}$ gives rise to a Kleisli triple $(Q, \eta, \hat{\cdot})$ (hence a monad) with $Q$ an endofunction on $\text{Ob}(\mathbb{S})$, $\eta$ a family of sketch morphisms embedding a sketch in its $Q$-image, and $\hat{\cdot}_Q: \mathbb{S}(V, Q(S)) \times \mathbb{S}(S, Q(T)) \rightarrow \mathbb{S}(V, Q(T))$ the Kleisli composition for any triple $V, S, T \in \text{Ob}(\mathbb{S})$.

5. Conclusion and Future work

In this extended abstract we presented central definitions and results about diagram logic and diagram algebra. The accompanying technical report [DS20] provides proofs, and an outlook of our future work plans. The main point is to extend the results currently obtained for one-layer operation signatures for signatures with pre- and postconditions to allow for diagrammatic reasoning.
REFERENCES


Appendix A. Using views in software engineering: Example

Figure 5 shows a simple example of views in software modeling. The source is specified by an UML class diagram [RJB04] describing a part of a university information structure in an almost self-explanatory way: Boxes refer to classes of objects Person, Course etc. and show their attributes (name, workload, etc.), arrows denote associations to be instantiated by mappings (relations or functions), and expressions \([m..n] \text{\(m,n \in \{0,1,2,\ldots,*\}\)}\) near associations ends are called multiplicities and constrain the type of relation/function. By default (and thus omitted in Fig. 5) every association has a target multiplicity of \([1..1]\) (i.e. exactly one) and a source multiplicity of \([0..*]\) (i.e. arbitrary), which translates to function-semantics (left total and right unique). With different multiplicities, one can specify different relations, e.g. the source multiplicity \([1..*]\) and target multiplicity \([2..*]\) at the (derived) association student\(^{-1}\).course enforce a surjective multi-valued function that maps a student object to at least two courses. Furthermore, there are different arrow-types, e.g. the arrows between Employee and UniversityRole, and Student and UniversityRole show the inheritance (subset) relationship between classes. Thus both Employee and Student inherit the person association from UniversityRole. Constraints and inheritance are the most prominent examples of builtin-constraints in UML, which can express domain specific rules, e.g. every student must at least subscribe to two courses and every course must have at least one student subscribed to it. However, there may be domain-specific rules that cannot be expressed with builtin constraints but instead require attached constraints, which are commonly formulated in the Object Constraint Language (OCL) [WK99]. For example, the

![Sample UML diagram](image-url)
requirement that student teaching assistants are not allowed to earn more than 15,000.00 a year due to tax regulations would be formalized as shown in Listing 1 and indicated in Fig. 5 with a note-symbol attached to class Employee. We can express this constraint in an element-free way by taking suitable pullbacks along inheritance monics and the monic \( \{15000\} \to \text{Real} \), but it would be a bulky and opaque construction for a practical engineer. Instead, in our generalized sketch framework, we consider such an OCL constraint as a formal constraint declaration \( c \) whose scope \( \text{diagr}(c) \) consists of the inheritance triangle (i.e., the corresponding cospan) and attribute \( \text{salary} \) (considered as an arrow from \( \text{Employee} \) to \( \text{Real} \)). Semantics of the constraints is the Boolean-valued procedure specified in the Listing, which classifies any configuration of sets and mappings over the scope as either valid or invalid.

**Listing 1. OCL constraint**

<table>
<thead>
<tr>
<th>context Employee inv TAPayment:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student.allInstances() ( \to \exists ) (s</td>
</tr>
<tr>
<td>implies self.salary &lt;= 15000.00</td>
</tr>
</tbody>
</table>

Note special names of the elements \( \text{student}^{-1} \) and \( \text{student}^{-1}.\text{course} \) (in blue colour and dashed), which declare these elements as terms built with the corresponding queries (operations): Arrow Inversion (say, query \( q_1 \)), and Arrow Composition (query \( q_2 \)). These queries are diagrammatic operations with obvious arities. The resulting arrow is used by the view schema \( V \) (to the right of the dashed vertical bar), which names this arrow as \( \text{takes} \). Formally, we express this as a mapping \( v: Q_v(S) \to V \) where \( Q_v = \{q_1, q_2\} \) and \( Q_v(S) \) is the full (black and blue) model (sketch) shown in the figure, whereas the source model is the (black) subsketch \( S \) not including the derived arrows.

Importantly, the view definition mapping \( v \) is compatible with constraints: multiplicities for \( \text{takes} \) in \( V \) correspond to those in \( Q_v(S) \), which makes \( v \) a sketch morphism. Then any valid model of \( S \), ie, a sketch morphism \( M: S \to \text{Set} \), when, first, being extended to \( M: Q_v(S) \to \text{Set} \), and then composed with the view definition mapping \( v \circ M: V \to \text{Set} \), will be a correct sketch morphism and hence a valid model of \( V \) satisfying all constraints declared in \( V \). Thus, the category hosting our artifacts is to be a category of sketches rather than graphs.