Institutions for SQL Database Schemas and Datasets

Martin Glauer, Till Mossakowski

To cite this version:

Martin Glauer, Till Mossakowski. Institutions for SQL Database Schemas and Datasets. 24th International Workshop on Algebraic Development Techniques (WADT), Jul 2018, Egham, United Kingdom. pp.67-86, 10.1007/978-3-030-23220-7_4 . hal-02364575

HAL Id: hal-02364575
https://inria.hal.science/hal-02364575v1

Submitted on 15 Nov 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Institutions for SQL Database Schemas and Datasets

Martin Glauer and Till Mossakowski

Otto von Guericke University Magdeburg, Germany
{martin.glauer,till.mossakowski}@ovgu.de

Abstract. Databases and the query language SQL play a major role in modern applications. In this paper we present an institution-based formalisation of relational databases that uses structures close to those used in SQL. This is the essential difference to other category-theoretical formalisations of databases, which often depart quite far from the SQL standard. We also study SQL queries, using institutional monads, and prove cocompleteness and amalgamation results for the institution.

Keywords: databases, SQL, category theory, institutions, queries, derived signature morphisms

1 Introduction

Database techniques have a long tradition in computer science and build the foundation for numerous modern applications. The *Structured Query Language* (SQL) [11] forms the fundamental specification for most state-of-the-art query languages for many database systems.[1] Near to all applications that rely on databases use some query language that conforms to the SQL specification, e.g. MySQL or PostgreSQL. This central role made SQL an essential building block of modern database technology that can not be replaced by other languages easily. Thus, formal approaches towards databases should focus this existing standard rather than develop a rivalling query language.

The diversity and heterogeneity of data needed for one specific application raises the need for frictionless interaction and migration between different database schemas. The growing role of Linked Open Data underlines this need. As a result there this a vast landscape of formal approaches towards databases and data migration, including many category-theoretical approaches. The relational make-up of many database schemas makes it natural to model them as categories and yields the intuitive notion of a schema merge as pushouts of functors. In [22, 23, 21] tables are represented as categories and values and relations between tables are functors. The Functorial Query Language yields functionalities to query those structures. A similar approach towards schema integration was defined for the first time in an institutional setting in [1]. Institutions were

[1] For big data applications, NoSQL databases play an important role, but even there, also SQL databases are used.

defined in [10] as a framework to cover the vast, heterogeneous landscape of logical formalisms used in computer science. The benefits of a categorical or institutional formalisation are 1) better tools for database structuring, 2) database integration can be achieved via colimits and 3) the possibility of heterogeneous integration of databases with logical languages.

However, previous approaches to category-theoretic formalisation of databases do not well integrate with actual practice of SQL databases. The reasons are that the defined structures are not close to actual relational database structures, the logic is not three-valued as in SQL[2], data tables cannot contain duplicates, the operators are not exactly those of SQL, and more (see also [22] for differences between algebraic and relational databases). Therefore, in this work, we aim at formalising an SQL institution that comes as close as possible to the SQL standard. This means there should be a simple and obvious correspondence between SQL database schemas and theories in the institution. This is particularly important for tool support via the Heterogeneous tool set (Hets) [14], which provides a software interface for institutions. Only with an institution supporting SQL directly, Hets can read in standard SQL database schemas. If needed, these can then be translated to other institutions. The formalisation presented in this paper can be seen a first step towards institution based logical reasoning on relational databases and their heterogeneous integration with logical theories. For example, in [20], using institutions, (a simplified form of) a database schema is heterogeneously integrated with an ontology formulated in the description logic OWL, via a first-order theory playing the role of a bridge theory. Our work will enable the use of real-world SQL database schemas in such heterogeneous integration scenarios.

2 An Institution for Databases

There is a vast landscape of different logics with varying expressiveness, complexity and purpose. Yet, most logics define syntactical entities like signatures (vocabularies) and sentences, as well as concepts of model and satisfaction (of a sentence in a model). Goguen et al defined institutions in [10] as an overarching, abstract framework that covers all those concepts and thus integrates them into a common structure.

Definition 1. *An institution* $\mathcal{I} = (\boldsymbol{Sign}^{\mathcal{I}}, \boldsymbol{Sen}^{\mathcal{I}}, \boldsymbol{Mod}^{\mathcal{I}}, \models^{\mathcal{I}})$ *consists of*

(i) *a category of* signatures $\boldsymbol{Sign}^{\mathcal{I}}$ *and signature morphisms;*

(ii) *a* sentence functor $\boldsymbol{Sen}^{\mathcal{I}} : \boldsymbol{Sign}^{\mathcal{I}} \to \mathbf{Set}$ *(where* $\mathbf{Set}$ *is the category of sets), providing for each signature* Σ *a set of sentences* $\boldsymbol{Sen}^{\mathcal{I}}(\Sigma)$ *and for each signature morphism* $\sigma : \Sigma_1 \to \Sigma_2$ *a sentence translation map* $\boldsymbol{Sen}^{\mathcal{I}}(\sigma) : \boldsymbol{Sen}^{\mathcal{I}}(\Sigma_1) \to \boldsymbol{Sen}^{\mathcal{I}}(\Sigma_2)$ *(also written* $\sigma(-)$*);*

[2] This also means that the subtle difference in the semantics of integrity constraints and that of *where* conditions in queries cannot be captured. We capture it: see our discussion on designated truth values below.

(iii) *a contra-variant* model functor $\boldsymbol{Mod}^{\mathcal{I}} : (\boldsymbol{Sign}^{\mathcal{I}})^{\mathrm{op}} \to \mathbf{Cat}$ *(where* **Cat** *is the category of categories*[3]*), providing for each signature* Σ *a category of models* $\boldsymbol{Mod}^{\mathcal{I}}(\Sigma)$ *and for each signature morphism* $\sigma : \Sigma_1 \to \Sigma_2$ *a model reduct functor* $\boldsymbol{Mod}^{\mathcal{I}}(\sigma) : \boldsymbol{Sen}^{\mathcal{I}}(\Sigma_2) \to \boldsymbol{Sen}^{\mathcal{I}}(\Sigma_1)$ *(also written* $-|_\sigma$*); and*

(iv) *a family of* satisfaction relations $\models^{\mathcal{I}}_{\Sigma} \subseteq |\boldsymbol{Mod}^{\mathcal{I}}(\Sigma)| \times \boldsymbol{Sen}^{\mathcal{I}}(\Sigma)$ *indexed over* $\Sigma \in |\boldsymbol{Sign}^{\mathcal{I}}|$,

such that the following satisfaction condition *holds for every signature morphism* $\sigma : \Sigma \to \Sigma'$ *in* $\boldsymbol{Sign}^{\mathcal{I}}$*, every sentence* $\varphi \in \boldsymbol{Sen}^{\mathcal{I}}(\Sigma)$ *and for every* Σ'*-model* $M' \in |\boldsymbol{Mod}^{\mathcal{I}}(\Sigma')|$*:*

$$M'|_\sigma \models^{\mathcal{I}}_{\Sigma} \varphi \;\Leftrightarrow\; M' \models^{\mathcal{I}}_{\Sigma'} \sigma(\varphi) \ .$$

Definition 2. *A* semi-institution $(\boldsymbol{Sign}, \boldsymbol{Mod})$ *(called specification frame in [7]) consists of a signature category* $\boldsymbol{Sign}$ *and a model functor* $\boldsymbol{Mod}^{\mathcal{I}} : (\boldsymbol{Sign}^{\mathcal{I}})^{\mathrm{op}} \to \mathbf{Cat}$ *(that is, it is an institution without sentences and satisfaction relation).*

In this section we will define our institution for relational SQL databases. The general structure is as follows: Each object of the signature category defines the schema of a relational database (e.g. tables, their columns and sorts) as well as the building blocks for expressions (i.e. function and predicate symbols). Sentences represent constraints on these tables (e.g. foreign key constraints) that the data in the database must conform to. The data stored in the database are objects of the model category and morphisms amongst these models represent multiset inclusion. Finally, the satisfaction relation checks whether the data the database (i.e. a model) conforms to a specified constraint (i.e. a sentence). As a guidance all parts of the institution will be explained using the following example tables for persons and employees. Both tables feature a primary key that is an integer. The employee table contains a foreign key reference to the person table, with an according constraint (i.e. the `pid` of an employee always must be the `id` of a person). Moreover, the employee table contains a check constraint ensuring that the salary of an employee cannot negative.

Example 1.

Person

id:int	fname:Text	lname:Text

Employee

id:int	salary:int	pid:int

```
CREATE TABLE Person(
    id  Int,
    fname Text,
```

[3] Strictly speaking, **Cat** lives in a higher set-theoretic universe, such that it is not member of itself.

```
    lname Text,
    PRIMARY KEY (id));

CREATE TABLE Employee(
    id Int,
    salary Int,
    pid Int,
    PRIMARY KEY (id),
    FOREIGN KEY (pid) REFERENCES Person(id),
    CHECK (salary >= 0));
```

The database schema describes the general structure of a database. It includes information about all tables in the database, which columns belong to a specific table and their respective datatype. Information about the structure of a database is expressed in the *Data Definition Language* (DDL). Many basic components (e.g. datatypes, operators) of a database are fixed by the underlying database system. Similarly, the SQL Institution is parameterised over $(D\Sigma, DM)$ consisting of a *datatype signature* $D\Sigma$, which is a many-sorted first-order signature $D\Sigma = (\mathbb{S}^{D\Sigma}, \mathcal{F}_{D\Sigma})$ such that

1. there is a sort $bool \in \mathbb{S}^{D\Sigma}$
2. for each sort $s \in \mathbb{S}^{D\Sigma}$ there is a constant $null : s$, unary predicates $is_null : s$ and $is_not_null : s$, and a binary predicate $=_s$: $s \times s$.

and of a *datatype model* DM, which is a $D\Sigma$-model such that

1. $DM_{bool} = \{\mathsf{T}, \mathsf{F}, \mathsf{U}\}$. (Note that SQL builds upon a 3-valued logic in order do cope with the existence of *null*-values.)
2. All operations are *strict*, i.e. they return $null_{DM}$ (or U, in case of operations with result sort *bool*) if any of the operands is $null_{DM}$. The only exceptions to this rule are *is_null* and *is_not_null*.
3. is_null_{DM} is T for $null_{DM}$, and F otherwise.
4. $is_not_null_{DM}$ is F for $null_{DM}$, and T otherwise.
5. $a(=_s)_{DM} b = \begin{cases} \mathsf{T} & a = b \neq null_{DM} \\ \mathsf{F} & null_{DM} \neq a \neq b \neq null_{DM} \\ \mathsf{U} & otherwise \end{cases}$

Example 2. A many-sorted first order signature for MySQL-datatypes may consist of the following components:

```
logic CASL.FOL=
spec Mysql_datatypes =
  sorts Bit, Bool, Int, Float, Text, Blob ...
  ops 0 : Int; null : Int; null : Float; null: Text;
      ...
  ops __+__, __*__ : Int * Int -> Int
  ops is_null, is_not_null : Int -> Bool; ...
  ops __=__, __>=__ Int * Int -> Bool; ...
  ops empty : Text -> Bool; ...
```

An extension for the database PostgreSQL (which also features geometrical datatypes, for geo databases) might look as follows:

```
spec Postgresql_datatypes =
Mysql_datatypes then
  sorts Point, Line, Polygon  ...
```

Both signatures can be interpreted with a model in a standard way, following the SQL conventions for datatypes and their built-in operations.

For a given pair $(D\Sigma, DM)$ of a data type signature and a data type model, we now define an institution $\mathcal{SQL}(D\Sigma, DM)$ as follows.

A signature in $\mathcal{SQL}(D\Sigma, DM)$ defines the tables the database contains as well as the corresponding columns and their datatypes.

Definition 3 (Signatures). *An object Σ of the category of signatures* ***Sign*** *consists of:*

- *a set of table names (or short: tables) $\mathbb{T}^\Sigma$,*
- *for each table $t \in \mathbb{T}^\Sigma$, a set $col^\Sigma(t)$ of columns,*
- *a function $\tau(\cdot,\cdot) : \{(t,c) | t \in \mathbb{T}^\Sigma, c \in col^\Sigma(t)\} \to \mathbb{S}^{D\Sigma}$ assigning sorts (of the datatype signature) to columns in tables, and*
- *for each table $t \in \mathbb{T}^\Sigma$, a primary key $pk^\Sigma(t) \subseteq col^\Sigma(t)$. Note $pk^\Sigma(t) = \emptyset$ is possible, which expresses, in SQL terms, the absence of a primary key.*

We include primary keys into signatures (instead of considering them to be sentences) because for each table there may be at most one primary key.

Definition 4 (Signature Morphisms). *A signature morphism $\sigma : \Sigma \to \Sigma'$ in the category of signatures* ***Sign*** *consists of*

- *a table translation $\sigma_\mathbb{T} : \mathbb{T}^\Sigma \to \mathbb{T}^{\Sigma'}$, and*
- *a family of column translations $\langle \sigma_{col,t} : col(t) \to col'(\sigma_\mathbb{T}(t)) \rangle_{t \in \mathbb{T}^\Sigma}$,*

such that types of columns are preserved

$$\tau^{\Sigma'}(\sigma_\mathbb{T}(t), \sigma_{col,t}(c)) = \tau^\Sigma(t,c)$$

and for each table $t \in \mathbb{T}^\Sigma$, the primary key (if existing) is preserved

$$pk^{\Sigma'}(\sigma_\mathbb{T}(t)) = \sigma_{col,t}(pk^\Sigma(t)) \text{ if } pk^\Sigma(t) \neq \emptyset$$

Example 3. A signature Σ for the tables shown in Example 1 is as follows:

- $\mathbb{T}^\Sigma = \{\texttt{Employee}, \texttt{Person}\}$
- $\mathrm{col}^\Sigma(\texttt{Employee}) = \{\texttt{id}, \texttt{salary}, \texttt{pid}\}$
- $\mathrm{col}^\Sigma(\texttt{Person}) = \{\texttt{id}, \texttt{fname}, \texttt{lname}\}$
- $\tau^\Sigma(\texttt{Employee}, .) = \{\texttt{id} \mapsto \texttt{Int}, \texttt{salary} \mapsto \texttt{Int}, \texttt{pid} \mapsto \texttt{Int}\}$
- $\tau^\Sigma(\texttt{Person}, .) = \{\texttt{id} \mapsto \texttt{Int}, \texttt{fname} \mapsto \texttt{Text}, \texttt{lname} \mapsto \texttt{Text}\}$
- $pk^\Sigma(\texttt{Person}) = \{\texttt{id}\}$

– $pk^{\Sigma}(\texttt{Employee}) = \{\texttt{id}\}$

The database schema described above yields means to structure data in tables with little limitations regarding the actual content of the data. In order to ensure data integrity, the SQL defines several types of constraints that may be placed on a table. The sentences of the presented institution formulate the most common types of constraints (`NOT NULL`, `UNIQUE`, `FOREIGN KEY` and `CHECK`) specified in the SQL standard.

Definition 5 (Sentences). *Given a signature Σ, a sentence φ^t is a constraint on a table $t \in \mathbb{T}^{\Sigma}$ of one of the following forms:*

– *A non-null constraint $notnull(t.c)$ ($c \in col(t)$)*
– *A unique constraint $un(t.c_1, \ldots, t.c_n)$ ($c_1, \ldots, c_n \in col(t)$)*
– *A foreign-key constraint $fk((t.c_1, u.d_1), \ldots, (t.c_n, u.d_n))$ for some a table $u \in \mathbb{T}^{\Sigma}$ and $c_i \in col(t)$, $d_i \in col(u)$ for $i = 1, \ldots, n$*
– *A check constraint $ck_t(\varphi)$, where φ is a term of sort Bool over the signature $D\Sigma$ (extended with logical connectives $\neg$, $\vee$, $\wedge$ as operations on Bool) and variables from the set $Var(t) = \{t.c \mid t \in col(t)\}$.*

Definition 6 (Sentence Translation). *Given a signature morphism $\sigma : \Sigma \to \Sigma'$, a constraint φ^t on a table $t \in \mathbb{T}^{\Sigma}$ is translated along σ to $\sigma(\varphi^t)$ depending on its structure*

– $\sigma(un(t.c_1, \ldots, t.c_n)) = un(\sigma_{\mathbb{T}}(t).\sigma_{col,t}(c_1), \ldots, \sigma_{\mathbb{T}}(t).\sigma_{col,t}(c_n))$
– $\sigma(fk((t.c_1, u.d_1), \ldots, (t.c_n, u.d_n))) = fk(\ (\sigma_{\mathbb{T}}(t).\sigma_{col,t}(c_1), \sigma_{\mathbb{T}}(u).\sigma_{col,t}(d_1)), \ldots, (\sigma_{\mathbb{T}}(t).\sigma_{col,t}(c_n), \sigma_{\mathbb{T}}(u).\sigma_{col,t}(d_n)))$
– *$\sigma(ck_t(\varphi))$ is obtained by mapping all variable names along σ, i.e. the variable $c.t$ is replaced with $\sigma_{col,t}(c).\sigma_{\mathbb{T}}(t)$.*

Example 4. Primary keys are already part of the signature. Thus, there are only two constraints remaining in Example 1 that can be formulated as sentences:

– The foreign key linking Employees to Persons

$$fk(\texttt{(Employee.pid,Person.id)})$$

– The constraint that the salary of an employee must not be negative

$$ck_t(\texttt{Employee.salary >= 0})$$

The signature fixes the structural make-up of the database and its tables. A model represents the data that may be stored in a database w.r.t. to the existing tables, columns and their datatypes. Note that in the interpretation of a table in a model, the same row may appear several times. Thus, tables are interpreted as multisets (also called bags) of rows. The use of multisets may seem overly complicated, because normalised databases cannot have duplicate rows. However, note that not all databases are normalised, and moreover duplicate rows can easily occur within views generated by queries.

For a set X, let $\wp_{multi}(X)$ be the set of finite multisets over X. A finite multiset $B \in \wp_{multi}(X)$ can be understood as a map $B : X \to \mathbb{N}$ such that $B(x) > 0$ only for finitely many $x \in X$. For $x \in X$ and $B \in \wp_{multi}(X)$, let $B\#x$ be the multiplicity of x in B. We write $x \in B$ for the fact that $B\#x > 0$. Moreover, we use the abbreviation $\exists! x \in B.\, \varphi(x)$ for

$$\Big(\sum_{\varphi(x)} B\#x\Big) = 1$$

i.e. there exists a unique x in B satisfying $\varphi(x)$, and this x has multiplicity 1. Furthermore, addition on finite multiset is defined element-wise:

$$(B_1 + B_2)\#x = B_1\#x + B_2\#x.$$

Finally, inclusion of multisets is defined as $A \subseteq B$ iff for all $x \in X$, $A\#x \leq B\#x$.

Definition 7 (Models). *Given a signature Σ, an object of the model category $\boldsymbol{Mod}(\Sigma)$ is a function mapping tables to multisets of functions that map columns to elements of the datatype model, formally $M : t \in \mathbb{T}^\Sigma \to \wp_{multi}(c \in col(t) \to DM_{\tau(t,c)})$, such that for each $t \in \mathbb{T}^\Sigma$ with $pk^\Sigma(t) = \{c_1, \ldots, c_n\} \neq \emptyset$*

- *$r(c_i) \neq null_{DM}$ for each row $r \in M(t)$ and $i = 1, \ldots, n$, and*
- *for all rows $r_1 \in M(t)$,*

$$\exists! r_2 \in M(t).\, r_1(c_1) = r_2(c_1) \wedge \cdots \wedge r_1(c_n) = r_2(c_n)$$

Note that in presence of the satisfaction relation introduced below, these conditions could be rephrased as

- $M \models \{notnull(t.c_1), \ldots, notnull(t.c_n)\}$
- $M \models un(t.c_1, \ldots, t.c_n)$

Morphisms in the category of models represent inclusion of data multisets.

Definition 8 (Model Morphisms). *Given a signature Σ, the model category $\boldsymbol{Mod}(\Sigma)$ is a partially ordered set, where $M_1 \leq M_2$ iff for each $t \in \mathbb{T}^\Sigma$, $M_1(t) \subseteq M_2(t)$.*

Definition 9 (Model Reducts). *Given a signature morphism $\sigma : \Sigma \to \Sigma'$ and a Σ'-model M' and a table $t \in \mathbb{T}^\Sigma$, a row $r' : (c' \in col(\sigma_{\mathbb{T}}(t))) \to DM_{\tau(\sigma_{\mathbb{T}}(t),c')})$ can be reduced to a row $r'|_\sigma : (c \in col(t)) \to DM_{\tau(t,c)})$ by defining*

$$r'|_\sigma(c) = r'(\sigma_{col,t}(c)) \text{ for } c \in col^\Sigma(t).$$

Then $M'|_\sigma$, the reduction of M' against σ, is defined by

$$\forall t \in \mathbb{T}^\Sigma, M'|_\sigma(t)\#r = \sum_{r'|_\sigma = r} M'(\sigma_{\mathbb{T}}(t))\#r' \tag{1}$$

for all $r : (c \in col(t)) \to DM_{\tau(t,c)})$. Note that $M'|_\sigma(t)\#r = 0$ in case there is no $r' \in M'(\sigma_{\mathbb{T}}(t))$ with $r'|_\sigma = r$. From M satisfying the primary key and not-null constraints of Σ', it is straightforward to see that $M'|_\sigma$ satisfies those of Σ.

It is easily seen that reducts preserve the partial order on models; hence they are functorial.

$\vee$	T	U	F
T	T	T	T
U	T	U	U
F	T	U	F

$\wedge$	T	U	F
T	T	U	F
U	U	U	F
F	F	F	F

$\neg$	
T	F
U	U
F	T

Fig. 1. Boolean operations in SQL's 3-valued logic

Definition 10 (Satisfaction Relation). *For every $M \in \boldsymbol{Mod}(\Sigma)$ the satisfaction relation is defined depending on the structure the sentence:*

- $M \models_\Sigma \mathit{notnull}(t.c)$ *iff* $r(c) \neq \mathit{null}_{DM}$ *for each row* $r \in M(t)$
- $M \models_\Sigma \mathit{un}(t.c_1, \dots, t.c_n)$ *iff for all rows* $r_1 \in M(t)$,

$$\exists! r_2 \in M(t).\, r_1(c_1) = r_2(c_1) \wedge \dots \wedge r_1(c_n) = r_2(c_n)$$

- $M \models_\Sigma \mathit{fk}((t.c_1, u.d_1), \dots, (t.c_n, u.d_n))$ *iff for all* $r_1 \in M(t)$ *there is exactly one* $r_2 \in M(u)$ *such that* $r_1(c_i) = r_2(d_i)$ *for* $i = 1, ..., n$
- $M \models_\Sigma \mathit{ck}_t(\varphi)$ *iff for all* $r \in M(t)$, $[\![\varphi]\!]^M_{\nu(r)} \in \{\mathsf{T}, \mathsf{U}\}$, *where for* $r \in M(t)$, $\nu(r)$ *is the valuation defined by* $\nu(t.c) = r(c)$ *for all* $c \in \mathit{col}(t)$.

Here, $[\![\varphi]\!]^{DM}_\nu$ is the evaluation of the term φ in the model DM using a valuation ν of the free variables, defined inductively in a standard way, where Boolean connectives are interpreted as shown in Fig. 1. Note that since $[\![\varphi]\!]^{DM}_\nu$ is of sort Bool, $[\![\varphi]\!]^{DM}_\nu \in \{\mathsf{T}, \mathsf{F}, \mathsf{U}\}$.

While internally, our institution $\mathcal{SQL}(D\Sigma, DM)$ uses a three-valued logic for the semantics of integrity constraints, ultimately, it is a two valued logic. This is achieved by considering sentences evaluating to T or U (the designated truth values, in standard many-valued logic terminology [17]) to hold, and considering those evaluating to F as not to hold. (When eliminating U in the first place, one would obtain a different logic.) Further note that for conditions in queries, SQL takes T as the only designated truth value, see Sect. 3 below. This means that queries impose a stricter regime than integrity constraints, which is pragmatically motivated: missing data should not lead to violation of an integrity constraint, while in queries, all mentioned data should be present.

Proposition 1 (Satisfaction condition). *Let $\sigma : \Sigma \to \Sigma'$ be a signature morphism between two database signatures. For all Σ'-models M' and Σ-constraints φ the following holds:*

$$M'|_\sigma \models \varphi \quad \Leftrightarrow \quad M' \models \sigma(\varphi)$$

Proof. Let M' be a Σ'-model and φ a Σ-sentence. The satisfaction follows directly from the fact that model reducts do not alter the data contained in the tables, apart from altering table and column names:

case $\varphi = \mathit{notnull}(t.c)$**:**

$$\begin{aligned} M'|_\sigma \models \mathit{notnull}(t.c) &\Leftrightarrow \forall r \in M'|_\sigma(t) : r(c) \neq \mathit{null}_{DM} \\ &\overset{Eq.1}{\Leftrightarrow} \forall r' \in M'(\sigma_{\mathbb{T}}(t)) : r'|_\sigma(c) \neq \mathit{null}_{DM} \\ &\Leftrightarrow \forall r' \in M'(\sigma_{\mathbb{T}}(t)) : r'(\sigma_{\mathrm{col},t}(c)) \neq \mathit{null}_{DM} \\ &\Leftrightarrow M' \models \sigma(\mathit{notnull}(t.c)) \end{aligned}$$

case $\varphi = \mathit{un}(t.c_1, \dots, t.c_n)$**:**

$$\begin{aligned} & M'|_\sigma \models \mathit{un}(t.c_1, \dots, t.c_n) \\ \Leftrightarrow\; & \forall r_1 \in M'|_\sigma(t) : \exists! r_2 \in M'|_\sigma(t) : \textstyle\bigwedge_{i=1,\dots,n} r_1(c_i) = r_2(c_i) \\ \overset{Eq.1}{\Leftrightarrow}\; & \forall r_1 \in M'(\sigma_{\mathbb{T}}(t)) : \exists! r_2 \in M'(\sigma_{\mathbb{T}}(t)) : \textstyle\bigwedge_{i=1,\dots,n} r_1|_\sigma(c_i) = r_2|_\sigma(c_i) \\ \Leftrightarrow\; & \forall r_1 \in M'(\sigma_{\mathbb{T}}(t)) : \exists! r_2 \in M'(\sigma_{\mathbb{T}}(t)) : \textstyle\bigwedge_{i=1,\dots,n} r_1(\sigma_{\mathrm{col},t}(c_i)) \\ & \qquad\qquad = r_2(\sigma_{\mathrm{col},t}(c_i)) \\ \Leftrightarrow\; & M' \models \sigma(\mathit{un}(t.c_1, \dots, t.c_n)) \end{aligned}$$

case $\varphi = \mathit{fk}((t.c_1, u.d_1), \dots, (t.c_n, u.d_n))$**:**

$$\begin{aligned} & M'|_\sigma \models \mathit{fk}((t.c_1, u.d_1), \dots, (t.c_n, u.d_n)) \\ &\quad \Leftrightarrow \forall r \in M'|_\sigma(t) \exists! s \in M'|_\sigma(u) : \forall i \in \{1, ..., n\} : r(c_i) = s(d_i) \\ &\quad \overset{Eq.1}{\Leftrightarrow} \forall r \in M'(\sigma_{\mathbb{T}}(t)) \exists! s \in M'(\sigma_{\mathbb{T}}(u)) : \forall i \in \{1, ..., n\} : \\ &\qquad r|_\sigma(c_i) = s|_\sigma(d_i) \\ &\quad \Leftrightarrow \forall r \in M'(\sigma_{\mathbb{T}}(t)) \exists! s \in M'(\sigma_{\mathbb{T}}(u)) : \forall i \in \{1, ..., n\} : \\ &\qquad r(\sigma_{\mathrm{col},t}(c_i)) = s(\sigma_{\mathrm{col},u}(d_i)) \\ &\quad \Leftrightarrow M' \models \sigma(\mathit{fk}((t.c_1, u.d_1), \dots, (t.c_n, u.d_n)))) \end{aligned}$$

Concerning check constraints, one can prove by induction over φ that

$$[\![\varphi]\!]^{DM}_{\nu(r|_\sigma)} = [\![\sigma(\varphi)]\!]^{DM}_{\nu(r)}$$

From this, the satisfaction condition for check constraints follows using Eq. 1. $\square$

This completes the definition of the institution $\mathcal{SQL}(D\Sigma, DM)$.

3 Queries and Views

Our institution $\mathcal{SQL}(D\Sigma, DM)$ provides a formalisation of SQL database schemas as logical theories in that institution. Now an important feature of SQL is of course the *Data Query Language* (DQL). Queries cannot be directly represented as logical formulas in the institution, because as an answer, they deliver data and not just logical truth values. Therefore, formalisations for dealing with Prolog-style queries in an institutional setting usually use generalised substitutions [3, 5] or derived signature morphisms. The latter are a generalised version of signature morphisms that may map signature symbols not only to other such symbols, but also to complex terms and expressions [13, 19]. Then Prolog-style

queries can be formalised as open sentences and their answer substitutions as derived signature morphisms[4].

However, it turns out that queries in SQL are different from Prolog-style queries. Indeed, SQL queries do not provide answer substitutions that map variables to terms, and so they cannot naturally be extended to a translation of sentences either. This is because SQL queries have a rich syntax, which in general cannot be reflected in SQL sentences (constraints). Instead, SQL queries provide a mapping of databases, i.e. models (see also Table. 1). Compared of abstract substitutions [3, 5], the syntactic half of the structure is missing. This means we would need to use abstract *semi*-substitutions. However, we follow a slightly different path. Namely we use Kleisli morphisms in an institutional monad as an institution-independent formalisation of derived signature morphisms. Queries can then be formalised as such Kleisli morphisms, and materialised views (answer to queries) are reducts against these. Moreover, Kleisli composition gives a means to compose queries, which is also called query unfolding in database terminology [6].

	query	**answer**
Prolog-style	open sentence	derived signature morphism
database-style	derived signature morphism	reduct against derived signature morphism

Table 1. Different types of queries

The notion of institutional monad [13] has been introduced for formalising the notion of derived signature morphism in an arbitrary institution; we weaken it here to the notion of *semi*-institutional monad, based on the notion of institution semi-morphism. We first recall the latter. Institution semi-morphisms relate two institutions by relating their signatures and models, while sentences are not related. This is useful if the institutions are semantically related, but differ too much in their sentences to have a full institution morphism between them.

Definition 11 ([10]). *Given institutions $\mathcal{I}$ and $\mathcal{J}$, an* institution semi-morphism $\mu = (\Phi, \beta)\colon \mathcal{I} \longrightarrow \mathcal{J}$ *consists of*

- *a functor* $\Phi\colon \boldsymbol{Sign}^{\mathcal{I}} \longrightarrow \boldsymbol{Sign}^{\mathcal{J}}$ *and*
- *a natural transformation* $\beta\colon \boldsymbol{Mod}^{\mathcal{I}} \longrightarrow \Phi^{op}; \boldsymbol{Mod}^{\mathcal{J}}$.

Definition 12 ([5]). *Given two institution semi-morphisms $\mu_1, \mu_2\colon \mathcal{I} \longrightarrow \mathcal{J}$ with $\mu_i = (\Phi_i, \beta_i)$, a (discrete)* institution semi-morphism modification $\tau\colon \mu_1 \longrightarrow \mu_2$ *is a natural transformation $\tau\colon \Phi_1 \longrightarrow \Phi_2$ such that*

[4] This way of formalising queries also has been proposed in an informal annex within the DOL language standard [16].

$$
\begin{array}{ccc}
\Phi_1^{op}; \boldsymbol{Mod}^{\mathcal{J}} & \xleftarrow{\tau^{op} * \boldsymbol{Mod}^{\mathcal{J}}} & \Phi_2^{op}; \boldsymbol{Mod}^{\mathcal{J}} \\
\nwarrow^{\beta_1} & & \nearrow_{\beta_2} \\
& \boldsymbol{Mod}^{\mathcal{I}} &
\end{array}
$$

commutes.

Definition 13 (Semi-institutional monad). *Let $\mathcal{I}$ be an institution. A* ***semi-institutional monad*** (T, η, μ) *consists of*

- *An institution semi-morphism $T : \mathcal{I} \to \mathcal{I}$*
- *A institution semi-morphism modification $\eta : id \to T$ (the unit of the monad)*
- *A institution semi-morphism modification $\mu : TT \to T$ (the multiplication of the monad)*

such that the usual laws of a monad are satisfied:

$$
\begin{array}{ccc}
T & \xrightarrow{T\eta} & TT \\
\eta_T \downarrow & = & \downarrow \mu \\
TT & \xrightarrow[\mu]{} & T
\end{array}
\qquad\qquad
\begin{array}{ccc}
TTT & \xrightarrow{T\mu} & TT \\
\mu_T \downarrow & = & \downarrow \mu \\
TT & \xrightarrow[\mu]{} & T
\end{array}
$$

The semi-institutional monad for the SQL data query language can now be sketched as follows. The institution semi-morphism $(\Phi, \beta) : \mathcal{SQL}(D\Sigma, DM) \to \mathcal{SQL}(D\Sigma, DM)$ of the semi-institutional monad maps a signature Σ to $\Phi(\Sigma)$, which extends Σ with tables q, where q is an SQL query over Σ. The tables q are also called *views*, because the table resulting from a query represents a new view on the database. Resulting in such a view, an SQL query naturally has an associated set of (result) columns (=materialised view), this will act as q's set of columns in $\Phi(\Sigma)$. q will not have a primary key. β maps an SQL-model (aka database) M into the model $\beta_\Sigma(M)$ which interprets each query q as the multiset of solutions of q over M. The monad unit η_Σ is simply the inclusion of Σ into $\Phi(\Sigma)$. The monad multiplication μ_Σ maps queries over queries to normal SQL queries. This is straightforward, because SQL allows to insert queries in positions where tables can occur (after all, views are special forms of tables), which is a form of query composition.

Since SQL is a rich language, the detailed formalisation of the complete SQL query language along the lines sketched above is beyond the scope of this paper. Instead, we concentrate on one particular form of SQL query, involving select, where and join. In a system, information is often distributed amongst multiple tables in order to prevent redundancies. In order to retrieve distributed data multiple tables can be joined into a view, using a query. When employing this restriction on queries, the above sketched semi-institutional monad can be detailed as follows, where the monad action given by the institution semi-morphism is denoted as $T = (\Phi, \beta)$:

– The signature functor $\Phi : \mathbf{Sign}^{\mathcal{SQL}} \to \mathbf{Sign}^{\mathcal{SQL}}$ extends a signature Σ by tables of form

$$t_1 \times \cdots \times t_n|_{\varphi_1,\ldots,\varphi_k} \to c_1 \blacktriangleright e_1, \ldots, c_m \blacktriangleright e_m$$

written in SQL notation as

```
SELECT e1 AS c1, ..., em AS cm
FROM t1 JOIN ... JOIN tn
WHERE φ1 AND ... AND φk
```

where $t_1, \ldots, t_n \in \mathbb{T}^\Sigma$, $\varphi_1, \ldots, \varphi_k$ are check constraints with the columns of $t_1, \ldots, t_n$ as variables, $c_1, \ldots, c_m$ are new column names, and each c_i is attached with a λ-expression e_i

$$\lambda(t_1.c_1, \ldots, t_1.c_{n_1}, \ldots, t_n.c_1, \ldots, t_n.c_{n_m}).t$$

that specifies the computation rules for this column depending on all columns of the joined tables. The type of t is called result type of e_i.
The columns of these tables are given as

$$\mathrm{col}(t_1 \times \cdots \times t_n|_{\varphi_1,\ldots,\varphi_k} \to c_1 \blacktriangleright e_1, \ldots, c_m \blacktriangleright e_m) = \{c_1, \ldots, c_m\}.$$

Sorts of columns are set w.r.t to the column expressions:

$$\tau(t_1 \times \cdots \times t_n|_{\varphi_1,\ldots,\varphi_k} \to c_1 \blacktriangleright e_1, \ldots, c_m \blacktriangleright e_m, c_i) = s_i$$

where s_i is the result type of e_i.

– $\beta_\Sigma(M)$ preserves the models behaviour on tables from Σ, i.e. $\beta_\Sigma(M)(t) = M(t)$ for $t \in \mathbb{T}^\Sigma$. In order to construct the interpretation of the new tables in $\Phi(\Sigma)$, we define $e_i(r_1, \ldots, r_n)$ as the evaluation of an expression e_i ($i = 1, \ldots, m$) w.r.t. to rows $r_1, \ldots, r_n$, i.e.

$$e_i(r_1, \ldots, r_n) := (\lambda(c_1^1, \ldots, c_{m_1}^1, \ldots, c_1^n, \ldots, c_{m_n}^n).t)(r_1(col(t_1)), \ldots, r_n(col(t_n)))$$

Each of the constructed rows has to conform to the constraints specified in $\varphi_1, \ldots, \varphi_k$. This can be easily guaranteed by checking whether the model $M^{r_1,\ldots,r_m}$ satisfies the set of constraints $(\varphi_1, \ldots, \varphi_k)$ where $M^{r_1,\ldots,r_n}$ contains just the row r_i for each table t_i ($i = 1, \ldots, n$) and no data for other tables. For a table $t = t_1 \times \cdots \times t_n|_{\varphi_1,\ldots,\varphi_k} \to c_1 \blacktriangleright e_1, \ldots, c_m \blacktriangleright e_m$ we construct the dataset as the evaluations of the corresponding column expressions:

$$\begin{aligned} \beta_\Sigma(M)(t) = \{r|\ & r(c_i) = e_i(r_1, \ldots, r_n)(i = 1, \ldots, m), \\ & r_1 \in M(t_1), \ldots, r_n \in M(t_n), \\ & [\![\varphi_j]\!]_{\nu_{r(\varphi_j)}}^{M^{r_1,\ldots,r_n}} = \mathsf{T}\ (j = 1, \ldots, k)\} \end{aligned}$$

Here $\nu_{r(\varphi_j)}$ is the valuation determined by the unique row r_i in table t_i, where t_i is the table over which φ_j has been formulated. As stated above (after Def. 10), for constraints in queries, T is the only designated truth value.

- η_Σ is the inclusion
- μ_Σ maps tables by collapsing an (outer) query over (inner) queries into a simple query. The result is the outer query, modified by substituting λ-terms for the inner queries into the column variables in λ-terms and check constraints of the outer query, and adding check constraints of the inner queries.

Example 5.
The tables from Example 1. A query

```
SELECT Person.salary AS salary,
       Person.fname AS fname,
       Person.lname AS lname
FROM Person JOIN Employee
ON Employee.pid = Person.id
WHERE Employee.salary >= 10000;
```

would correspond to the table in the monad

$$\texttt{Person} \times \texttt{Employee}|_{\texttt{Employee.pid = Person.id,Employee.salary >= 10000}} \to \\ \texttt{salary} \blacktriangleright e_1, \texttt{fname} \blacktriangleright e_2, \texttt{lname} \blacktriangleright e_3$$

where

$$e_1 = \lambda\texttt{p_id}, \texttt{p_fname}, \texttt{p_lname}, \texttt{e_id}, \texttt{e_salary}, \texttt{e_pid}.\texttt{e_salary}$$
$$e_2 = \lambda\texttt{p_id}, \texttt{p_fname}, \texttt{p_lname}, \texttt{e_id}, \texttt{e_salary}, \texttt{e_pid}.\texttt{e_fname}$$
$$e_3 = \lambda\texttt{p_id}, \texttt{p_fname}, \texttt{p_lname}, \texttt{e_id}, \texttt{e_salary}, \texttt{e_pid}.\texttt{e_lname}$$

In the sequel, we work with an arbitrary but fixed semi-institutional monad (T, η, μ) over an institution $\mathcal{I}$ (with $T = (\Phi, \beta)$). We can define several derived notions:

Definition 14 (Kleisli semi-institution, adapted from [13]). *The Kleisli signature category has a objects $\mathcal{I}$-signatures, and its morphisms $\Sigma_1 \to \Sigma_2$ are $\mathcal{I}$-signature morphisms $\Sigma_1 \to \Phi(\Sigma_2)$. Kleisli composition is defined by*

$$\Sigma_1 \xrightarrow{\sigma_1} \Phi(\Sigma_2) \;;\; \Sigma_2 \xrightarrow{\sigma_2} \Phi(\Sigma_3) \;=\; \Sigma_1 \xrightarrow{\sigma_1} \Phi(\Sigma_2) \xrightarrow{\Phi(\sigma_2)} \Phi(\Phi(\Sigma_3)) \xrightarrow{\mu_{\Sigma_3}} \Phi(\Sigma_3)$$

Model categories are inherited from $\mathcal{I}$. Model reduct against $\sigma : \Sigma_1 \to \Phi(\Sigma_2)$ is given by $\beta_{\Sigma_2}; \boldsymbol{Mod}^{\mathcal{I}}(\sigma)$. This gives a semi-institution in the sense of Def. 2.

Due to the "semi"-nature of the monad, we do not have sentence translations along Kleisli signature morphisms. This reflects the fact that queries use a much more powerful language than constraints, and constraints are not closed under queries.

Queries are just Kleisli signature morphisms:

Definition 15 (Query). *A* query *is a signature morphism* $q : \Sigma^Q \to \Phi(\Sigma)$.

Here, Σ^Q typically is a "small" or "singleton" signature. In the case of $\mathcal{SQL}(D\Sigma, DM)$, Σ^Q will typically consist of a single table only, providing a name and column types for the result of the query. The query itself is then given by $q(\Sigma^Q)$, which singles out one particular query in the space $\Phi(\Sigma)$ of all Σ-queries. The semantics of a query q as above is given by the Kleisli reduct $\beta_\Sigma; \mathbf{Mod}^{\mathcal{I}}(q)$, mapping Σ-models (aka Σ-databases) to Σ^Q-models (aka materialised views of signature Σ^Q).

Two queries $q_1, q_2 : \Sigma^Q \to \Phi(\Sigma)$ are *equivalent* [6] if their semantics agree, i.e. $\beta_\Sigma; \mathbf{Mod}(q_1) = \beta_\Sigma; \mathbf{Mod}(q_2)$. q_1 is *contained in* q_2 if for each $M \in \mathbf{Mod}(\Sigma)$, there is a model monomorphism $\beta_\Sigma(M)|_{q_1} \to \beta_\Sigma(M)|_{q_2}$.[5]

Next, we consider composition of queries, also known as query unfolding, which is an important tool for data base integration [6]. Given two queries $q_1 : \Sigma_1^Q \to \Phi(\Sigma_1)$ and $q_2 : \Sigma_2^Q \to \Phi(\Sigma_2)$, we cannot compose them directly. Rather, we have (assuming the existence of signature coproducts) to assume that the first query is built over the second database enriched with the result of the second query, i.e. that $\Sigma_1 = \Sigma_2^Q + \Sigma_2$. Then we can pair the second query with η and take the Kleisli composition

$$\Sigma_1^Q \xrightarrow{q_1} \Phi(\Sigma_1) \; ; \; \Sigma_1 = \Sigma_2 + \Sigma_2^Q \xrightarrow{[\eta, q_2]} \Phi(\Sigma_2)$$

We can also obtain a semi-substitution from a query. Substitutions have been introduced in [3, 5]. They run between variable sets that are represented as signature morphisms (extending a base signature with some constants playing the role of variables). We weaken the notion as follows:

Definition 16. *Given two signature morphisms ("variable sets")* $\chi_1 : \Sigma \to \Sigma_1$ *and* $\chi_2 : \Sigma \to \Sigma_2$*, a* Σ-semi-substitution $\psi : \chi_1 \to \chi_2$ *is a functor*

$$\boldsymbol{Mod}(\psi) : \boldsymbol{Mod}(\Sigma_2) \to \boldsymbol{Mod}(\Sigma_1)$$

such that the following diagram commutes:

$$\begin{array}{ccc} \boldsymbol{Mod}(\Sigma_1) & \xleftarrow{\boldsymbol{Mod}(\psi)} & \boldsymbol{Mod}(\Sigma_2) \\ & {\scriptstyle \boldsymbol{Mod}(\chi_1)} \searrow \quad \swarrow {\scriptstyle \boldsymbol{Mod}(\chi_2)} & \\ & \boldsymbol{Mod}(\Sigma) & \end{array}$$

For a query $q : \Sigma^Q \to \Phi(\Sigma)$, we consider the coproduct injection $\iota_1 : \Sigma \to \Sigma + \Sigma^Q$ as "variable set", and define the semi-substitution $\psi : \iota_1 \to 1_\Sigma$ via

$$\mathbf{Mod}(\psi) = \beta_\Sigma; \mathbf{Mod}^{\mathcal{I}}(\Sigma + \Sigma^Q \xrightarrow{[\eta_\Sigma, q]} \Phi(\Sigma)) \; .$$

[5] For $\mathcal{SQL}(D\Sigma, DM)$ this means multiset inclusion; and query equivalence is the same as containment in both directions.

Commutativity of the above diagram for semi-substitutions is ensured by the modification property of η (see Def. 12).

Another view on (general) Kleisli morphisms is that they correspond to (complex) schema mappings [22], because they map table names to queries. This means that we can compose schema mappings, and also (via the Kleisli model functor) compute their semantics.

4 Database Integration via Colimits

The formalisation of SQL databases as an institution allows the use of colimits for database integration.

Proposition 2. *Consider finite diagram $D : I \to \mathbf{Sign}^{\mathcal{SQL}}$, where for any span $j \xleftarrow{m} i \xrightarrow{n} k$ in I with $j \neq k$ and any $t \in \mathbb{T}^{D(i)}$, the following holds: if $D(m)_{\mathbb{T}}(t)$ and $D(n)_{\mathbb{T}}(t)$ both have a primary key, then so does t. Then D has a colimit.*

Proof. Let a finite diagram $D : I \to \mathbf{Sign}^{\mathcal{SQL}}$ with the above property be given. We construct its colimit signature Σ together with colimit injections $(\mu_i : D(i) \to \Sigma)_{i\in|I|}$ as follows. Let $D_{\mathbb{T}}$ be the projection of D to tables and table translations. Then let $\mathbb{T}^{\Sigma}$ be the colimit of $D_{\mathbb{T}}$ (in **Set**), and let $(\mu_i)_{\mathbb{T}} : \mathbb{T}^{D(i)} \to \mathbb{T}^{\Sigma}$ (for $i \in |I|$) be the colimit injections. This gives us the table part of the colimit. Concerning the column part, let $\mathbb{S}^{D\Sigma}$-**Set** be the category of $\mathbb{S}^{D\Sigma}$-sorted sets, which is given by the comma category $\mathbf{Set} \downarrow \mathbb{S}^{D\Sigma}$. For a table $t \in \mathbb{T}^{\Sigma}$, we construct a diagram of $\mathbb{S}^{D\Sigma}$-sorted column sets as follows. The index category I_t has objects

$$|I_t| = \{(i, u) \mid u \in \mathbb{T}^{D(i)}, (\mu_i)_{\mathbb{T}}(u) = t\},$$

i.e. the set of tables that are mapped (by some colimit injection) to t. I_t has a morphism $m : (i, u_1) \to (j, u_2)$ whenever $m : i \to j \in I$ and $D(m)_{\mathbb{T}}(u_1) = u_2$. The diagram functor $D_t : I_t \to \mathbb{S}^{D\Sigma}$-**Set** acts on objects as

$$D_t(i, u) = \mathrm{col}^{D(i)}(u),$$

which is $\mathbb{S}^{D\Sigma}$-sorted using $\tau^{D(i)}(u, \cdot)$, and on morphisms as

$$D_t(m : (i, u_1) \to (j, u_2)) = D(m)_{\mathrm{col},u_1} : \mathrm{col}^{D(i)}(u_1) \to \mathrm{col}^{D(j)}(u_2)$$

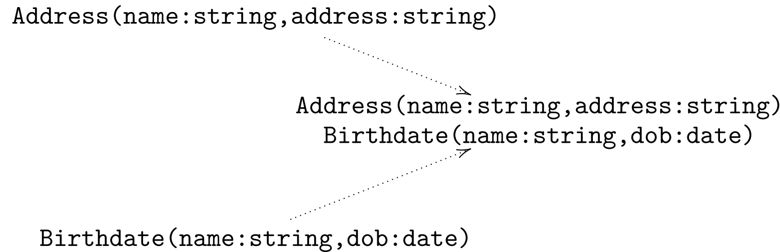
Let $(C_t, (\theta^t_{(i,u)})_{(i,u)\in|I_t|})$ be the colimit of D_t in $\mathbb{S}^{D\Sigma}$-**Set** (recall that colimits in comma categories are constructed component-wise). Then C_t is the sorted set of columns of table t in Σ. The column translation of $\mu_i : D(i) \to \Sigma$ is given by

$$(\mu_i)_{\mathrm{col},u} = \theta^{(\mu_i)_{\mathbb{T}}(u)}_{(i,u)} \qquad (u \in \mathbb{T}^{D(i)}).$$

Finally, the primary key of $C(t)$ is determined as $(\mu_i)_{\mathrm{col},u}(pk^{D(i)}(u))$ for any $(i, u) \in |I_t|$ with $pk^{D(i)}(u) \neq \emptyset$. Since any two such (i, u) are connected via a

zigzag path in I_t, the assumption about spans in I together with preservation of primary keys along signature morphisms ensures that this is independent of the choice of (i, u). This completes the construction of the colimit $(\Sigma, (\mu_i : D(i) \to \Sigma)_{i \in |I|})$. Its universal property follows from the universal properties of the colimits involved in its construction. □

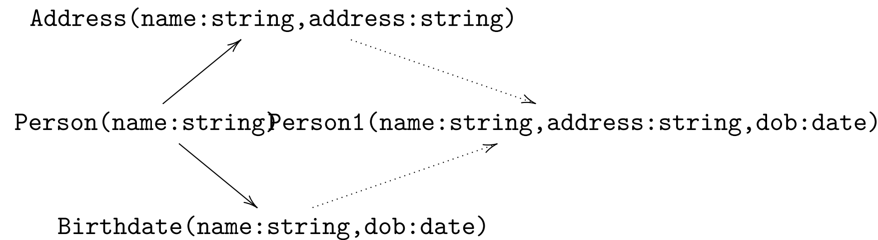
Example 6. Coproducts put database tables side by side:



Here, signature morphisms are inclusions.

An integration of tables can be achieved using pushouts:

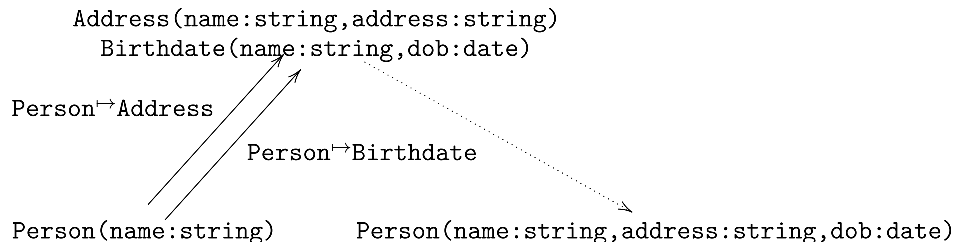
Example 7.



Here, signature morphisms map tables in the unique way and leave columns as they are. If table `Person` has primary key name, then this is also the primary key in the colimit. However, if `Person` does not have a primary key, but `Birthdate` and `Address` have one consisting of both of their respective columns, then the diagram does not have a colimit, because its primary key cannot be determined.

Since tables in one signature do not interact, the same effect can also be achieved with a coequaliser

Example 8.



Here, signature morphisms leave columns as they are. For table `Person`, at the table level, we have the coequaliser (in **Set**)

$$\texttt{Person} \rightrightarrows \texttt{Address, Birthdate} \dashrightarrow \texttt{Person}$$

At the column level, we obtain a pushout

```
              name:string,address:string
             ↗                          ↘
name:string          name:string,address:string,dob:date
             ↘                          ↗
              name:string,dob:date
```

So the coequaliser involves a coequaliser at the level of tables plus a pushout at the level of columns.

The amalgamation property is a major technical assumption in the study of specification semantics [18] and is important in many respects. For example, it allows the computation of normal forms for specifications [2], and it is a prerequisite for good behaviour w.r.t. parameterization [8] and conservative extensions [4] and for soundness of proof systems for structured specification [15]. Intuitively, amalgamation of databases is just a rearrangement of the (partially shared) tables.

Definition 17. *A cocone for a diagram in* $\textbf{\textit{Sign}}^{\mathcal{I}}$ *is called* (weakly) amalgamable *if it is mapped to a (Weak) limit under* $\textbf{\textit{Mod}}^{\mathcal{I}}$*. This can be characterised in more elementary terms as follows: Given a diagram* $D\colon I \longrightarrow \textbf{\textit{Sign}}^{\mathcal{I}}$*, a family of models* $(M_i)_{i\in|I|}$ *is called* D-consistent *if* $M_j|_{D(m)} = M_i$ *for each* $m\colon i \longrightarrow j \in I$*. Then a cocone* $(\Sigma, (\mu_i)_{i\in|I|})$ *over the diagram in* $D\colon I \longrightarrow \textbf{\textit{Sign}}^{I}$ *is weakly amalgamable if for each* D*-consistent family of models* $(M_i)_{i\in|I|}$*, there is a* Σ*-model* M *with* $M|_{\mu_i} = M_i$ $(i \in |I|)$ *(and moreover, an analogous condition holds for model morphisms). In case* M *is unique with these properties, the cocone is amalgamable.*

An institution $\mathcal{I}$ *admits* (weak) finite amalgamation *if all finite colimit cocones are (weakly) amalgamable.*

Proposition 3. *For colimit cocones as in Prop. 2,* $\mathcal{SQL}(D\Sigma, DM)$ *admits weak finite amalgamation, and if all diagram signatures have a primary key, then also finite amalgamation.*

Proof. Let $(\Sigma, (\mu_i)_{i\in|I|})$ be a colimit cocone as in Prop. 2. Moreover, let $(M_i)_{i\in|I|}$ a D-consistent family of models. We need to define its amalgamation $M \in \mathbf{Mod}^{\mathcal{SQL}}(\Sigma)$. Given $t \in \mathbb{T}^{\Sigma}$, consider the diagram $D_t : I_t \to \mathbb{S}^{D\Sigma}$-**Set** defined in the proof of Prop. 2. Recall that $\mathrm{col}^{\Sigma}(t)$ is the colimit of this diagram. The carrier sets of the model DM form an $\mathbb{S}^{D\Sigma}$-sorted set $|DM|$, and any row is

an $\mathbb{S}^{D\Sigma}$-sorted map into $|DM|$. Hence, any compatible family of rows $(r_{(i,u)} \in M_i(u))_{(i,u)\in|I_t|}$ is a cocone for D_t, and by the colimit property of $\mathrm{col}^{\Sigma}(t)$, it can be amalgamated to a row $r : (c \in \mathrm{col}(t)) \to DM_{\tau(t,c)}$. If all diagram signatures have a primary key, then let $M(t)\#r = 1$ for any such r, and M is the unique amalgamation. If not, then proceed with the following algorithm (working on a copy of $(M_i)_{i\in|I|}$ that is modified):

1. initially, set all multiplicities $M(t)\#r$ to 0.
2. Choose a compatible family of rows $(r_{(i,u)} \in M_i(u))_{(i,u)\in|I_t|}$, and for its amalgamation r, increase $M(t)\#r$ by one.
3. Decrease the multiplicities $M_i(u)\#r_{(i,u)}$ by one.
4. Repeat steps 2 and 3 until all multiplicities $M_i(u)\#r_{(i,u)}$ are zero. By Eq. 1, all $M_i(u)\#r_{(i,u)}$ will reach zero simultaneously. □

By very general results [10, 19], Props. 2 and 3 carry over from signatures to presentations, i.e. signatures equipped with axioms, e.g. foreign key constraints.

5 Conclusion

We presented an institutional approach towards a formalisation of relational databases and their integration via colimits and amalgamation. Building on a foundation of a many-sorted first-order structure, the signatures and sentences reflect SQL's Data Description Language, including tables, their columns and various constraints. The models of this institution capture the data that can be stored in those tables. Queries on a database can be formalised as a semi-institutional monad. Syntactic and semantic components of this institution were designed to closely follow those used in SQL, which allows an easy application of this new formalisation to existing database-related tools.

Many other categorical approaches define a new query language (e.g. FQL [24]). This limits the frictionless application of those approaches in real-world scenarios, as SQL is used in most software solutions that involve database communication. Therefore, we designed the presented institutional formalisation of databases to closely reflect the existing structures from the SQL specification.

Many open questions remain, e.g. whether model reducts have left and right adjoints as in [22], or whether colimits can be expressed as queries. Also, the existence of colimits in the Kleisli category should be studied, because these correspond to database integrations using queries. Note that by the results of [13], these colimits do not always exist.

Future work should also investigate the manipulation of data by insert and delete statements. Model morphisms could be used to capture a structure similar to those used in a categorical theory of patches [12], which may yield a logical framework for version tracking of databases.

Operations that involve multiple tables are present in most database-related application. The introduction of a suitable institutional semi-monad (and its Kleisli morphisms) enhances the presented institution for databases with functionalities such as inner joins that combine multiple tables. Yet, the current

formalisation allows only inner joins. Full formalisations of all different types of joins (e.g. left join, outer join) and other principles (e.g. aggregations, unions) are yet to be done. The introduction of other join types can be achieved by introducing multiple join operators with appropriate model interpretations. A similar approach could be chosen for unions. Aggregations will pose a tougher problem. They require an adaption of the notion of reduct of a Kleisli morphism, because a single row in a query result would stem from multiple rows of a single base table.

Institutions were defined as a framework that allows integration of different logics. Similarly, one may define institution comorphisms [9] to different logics (including logics for algebraic databases) and thus open the doors for the vast and powerful landscape of logic reasoning tools. The Heterogeneous tool set (Hets) [14] offers functionalities for reasoning and proving not only across logics, but also integrates languages like UML. The integration of the SQL-institution into this tool set can be a step towards a logics-founded approach for model-driven development for databases with automated checks for coherence. Integration of queries could be done by representing Kleisli morphisms[6], implementing their composition as well as checks for query containment and equivalence.

References

1. Alagić, S., Bernstein, P.A.: A model theory for generic schema management. In: International Workshop on Database Programming Languages. pp. 228–246. Springer (2001)
2. Borzyszkowski, T.: Generalized interpolation in CASL. Information Processing Letters **76/1-2**, 19–24 (2000)
3. Diaconescu, R.: Herbrand theorems in arbitrary institutions. Information Processing Letters **90**, 29–37 (2004)
4. Diaconescu, R., Goguen, J., Stefaneas, P.: Logical support for modularisation. In: Huet, G., Plotkin, G. (eds.) Proceedings of a Workshop on Logical Frameworks (1991)
5. Diaconescu, R.: Institution-independent Model Theory. Birkhäuser (2008)
6. Doan, A., Halevy, A.Y., Ives, Z.G.: Principles of Data Integration. Morgan Kaufmann (2012), http://research.cs.wisc.edu/dibook/
7. Ehrig, H., Große-Rhode, M.: Functorial theory of parameterized specifications in a general specification framework. Theoretical Computer Science **135**, 221–266 (1994)
8. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 2. Springer (1990)
9. Goguen, J., Roşu, G.: Institution morphisms. Formal aspects of computing **13**, 274–307 (2002)
10. Goguen, J.A., Burstall, R.M.: Institutions: Abstract model theory for specification and programming. Journal of the ACM (JACM) **39**(1), 95–146 (1992)
11. Melton, J.: ISO/IEC 9075-2: 2003 (SQL/foundation). ISO standard (2003)
12. Mimram, S., Di Giusto, C.: A categorical theory of patches. Electronic notes in theoretical computer science **298**, 283–307 (2013)

[6] Note that the signatures $\Phi(\Sigma)$ are generally infinite and cannot be easily represented, but for a Kleisli morphism, only the mapped symbols need to be represented.

13. Mossakowski, T., Krumnack, U., Maibaum, T.: What is a derived signature morphism? In: Codescu, M., Diaconescu, R., Tutu, I. (eds.) Recent Trends in Algebraic Development Techniques - 22nd International Workshop, WADT 2014, Sinaia, Romania, September 4-7, 2014, Revised Selected Papers. Lecture Notes in Computer Science, vol. 9463, pp. 90–109. Springer (2014). https://doi.org/10.1007/978-3-319-28114-8_6, http://dx.doi.org/10.1007/978-3-319-28114-8_6
14. Mossakowski, T., Maeder, C., Lüttich, K.: The Heterogeneous Tool Set. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. Lecture Notes in Computer Science, vol. 4424, pp. 519–522. Springer-Verlag Heidelberg (2007)
15. Mossakowski, T., Tarlecki, A.: A relatively complete calculus for structured heterogeneous specifications. In: Muscholl, A. (ed.) Foundations of Software Science and Computation Structures - 17th International Conference, FOSSACS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings. Lecture Notes in Computer Science, vol. 8412, pp. 441–456. Springer (2014). https://doi.org/10.1007/978-3-642-54830-7_29, https://doi.org/10.1007/978-3-642-54830-7_29
16. Object Management Group: The distributed ontology, modeling, and specification language (DOL) (2018), oMG standard available at http://www.omg.org/spec/DOL/
17. Rosser, J.B., Turquette, A.: Many-valued logics. North-Holland (1952)
18. Sannella, D., Tarlecki, A.: Specifications in an arbitrary institution. Information and Computation **76**, 165–210 (1988)
19. Sannella, D., Tarlecki, A.: Foundations of Algebraic Specification and Formal Software Development. Monographs in Theoretical Computer Science. An EATCS Series, Springer (2012). https://doi.org/10.1007/978-3-642-17336-3, https://doi.org/10.1007/978-3-642-17336-3
20. Schorlemmer, W.M., Kalfoglou, Y.: Institutionalising ontology-based semantic integration. Applied Ontology **3**(3), 131–150 (2008). https://doi.org/10.3233/AO-2008-0041, https://doi.org/10.3233/AO-2008-0041
21. Schultz, P., Spivak, D., Vasilakopoulou, C., Wisnesky, R.: Algebraic databases. Theory and Applications of Categories **32**(16), 547–619. (2017)
22. Schultz, P., Spivak, D.I., Wisnesky, R.: Algebraic model management: A survey. In: James, P., Roggenbach, M. (eds.) Recent Trends in Algebraic Development Techniques - 23rd IFIP WG 1.3 International Workshop, WADT 2016, Gregynog, UK, September 21-24, 2016, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10644, pp. 56–69. Springer (2016). https://doi.org/10.1007/978-3-319-72044-9_5, https://doi.org/10.1007/978-3-319-72044-9_5
23. Schultz, P., Wisnesky, R.: Algebraic data integration. J. Funct. Program. **27**, e24 (2017). https://doi.org/10.1017/S0956796817000168, https://doi.org/10.1017/S0956796817000168
24. Spivak, D.I., Wisnesky, R.: Relational foundations for functorial data migration. In: Proceedings of the 15th Symposium on Database Programming Languages. pp. 21–28. ACM (2015)