Data, Schema, Ontology and Logic Integration

Joseph A. Goguen

University of California, San Diego
Dept. Computer Science & Engineering

Abstract. This paper gives a general definition of a "kind of schema" (often called a "meta-model" in the literature, but here called a "species") along with general definitions for the schemas of a species, and for the databases, constraints, and queries over a given schema of a species. This leads naturally to a general theory of data translation and integration over arbitrary schemas of arbitrary species, based on schema morphisms, and to a similar general theory of ontology translation and integration over arbitrary logics. Institutions provide a general notion of logic, and Grothendieck flattening provides a general tool for integrating heterogeneous schemas, species and logics, as well as theories, such as ontologies, over different logics. Many examples of our novel concepts are included, some rather detailed. An initial section introduces data integration and ontologies for readers who are not specialists, with some emphasis on challenges. A brief review of universal algebra is also given, though some familiarity with category theory is assumed in later sections.

1 Introduction and Motivation

Data translation (also called "data exchange") and data integration have emerged as important challenges in the early twenty-first century. The rise of inexpensive storage media, data warehousing, and especially the web, have made vast amounts of data available. But it can be difficult to find what you want and correctly process it to get what you need. Reasons for this include highly variable data formats (e.g., spreadsheets, relational databases, informally formatted files, XML files, object oriented databases, and more) and highly variable data quality (e.g., entries may be incomplete, corrupted, or inconsistent, and there may be little or no meta-data to document either format or meaning). If all documents had associated schemas (also called data models) that accurately described their structure, and if fully automatic schema translation and integration were possible, then several important problems could be solved at a purely syntactic level [5]; however, these two assumptions are far from true. Moreover, format is only a small part of the difficulty, most of which is semantic and pragmatic. The growing popularity of XML will make some things easier, but cannot solve the basic problems, which are not only technical, but also social, as shown by difficulties with implementing the semantic web vision [4], as well as workflows which automate processing the enormous datasets that are increasingly common in astrophysics, proteomics, high energy physics, ecology, agriculture, pharmacology, e-business, geology, and many other areas.

Ontologies have been proposed as a solution; these are not metaphysical assertions about basic world substances made by philosophers, but rather are terminological systems, items from which can be attached to e-documents. They cannot capture real world semantics, but only logical relations between predicates, such as that all humans are mammals; the actual meanings of "human" and "mammal" remain unformalized. Moreover, a given domain may have several competing ontologies, each in some ways incomplete and/or ambiguous, and potentially written in different ontology languages, which in turn may be based upon different logical systems. OWL and RDF are currently most prominent, but others include Ontologic, $\mathcal{ALC}$, KIF, KL-ONE, XSB, Flora, and OIL; specialized ontology languages, e.g., Ecolingua and EML for ecology, tend not to have formal semantics. A sociological and philosophical discussion of limitations of ontologies is given in [17]. It follows from all this that human involvement cannot be entirely eliminated, and moreover, that data integration may require not just schema and ontology integration, but also ontology language integration, and even ontology logic integration, in such a way that semantics is respected throughout the entire "integration stack," from actual datasets or e-documents, through schemas, schema species, and ontologies, up to ontology logics.

It is useful (e.g., in discussing tool support) to distinguish four levels of support for data manipulation, for which we will use the following terminology:

1. **Schema matching** tools produce a relation between the node sets of two schemas; nodes may be elements, attributes, paths, or some combination of two or all three of these. Most current tools work at this level, and therefore rely on other tools, perhaps just a text editor, to provide the additional information needed to support data translation and integration.
2. **Schema mapping** tools provide enough information to generate a view that can be used for data translation. We use the term **schema morphism** for a mapping that produces *correct* data translation, noting that the ultimate criterion for correctness is satisfaction of the user. The language of the tool need not be the same as that of the views that it generates; indeed, in the best cases, the tool is GUI-based rather than text based like SQL or XQuery.
3. **Data integration tools** support translation of data from multiple sources and its subsequent integration to answer queries over a global schema.
4. **Heterogeneous data integration tools** provide translation and integration for data sources over schemas of different species.

A major obstacle to research in this area is that, although several particular species of schema are well defined (such as XML Schema), we do not have a general definition for the basic concept of schema species; similarly, we do not have a general definition for the concept of ontology language; and more significantly, we do not understand the various kinds of morphisms that are needed to support translations among such objects. Without such understandings, it is impossible to build effective tools to support heterogeneous data, schema and ontology integration. This paper is intended to fill these gaps, while also providing some technological background for those not already familiar with this area. In addition, it briefly discusses practical tools to support translation and integration of data at various levels of the integration stack. Institutions and their morphisms (these notions are explained in Section 4) are promising for understanding the higher levels of this stack, clarifying some problems, and suggesting sound solutions.

Section 2 provides informal background on databases, schemas, schema mappings, ontologies, workflows, and their roles in data integration; computer scientists may wish to skip this. Section 3 describes our approach to schemas, and their species, databases, constraints and morphisms. Section 3.1 is a brief introduction to algebraic specification theory, which is heavily used in the rest of the paper. Section 3.7 briefly describes the SCIA tool, which implements our theory for certain schema species.

Section 4 describes our approach to heterogeneous data and ontology integration, which uses institutions, institution morphisms, and the Grothendieck flattening construction to formalize the integration stack; some category theory is used here and in Section 3.6; categorical background can be found in [34, 37] and many other sources. The category theoretic view that morphisms are often more important than objects will be found very consistent with the subject of this paper. Section 4.2 briefly discusses the information flow (in the technical sense of Barwise and Seligman [2]) approach to integration, and Section 4.3 discusses ways to use ontologies for data integration. The many definitions in this paper are important as the basis of tools to translate and integrate data over schemas of different species, while the relatively few results establish properties needed for such applications; some results are designated "Theorem" despite having rather easy proofs, due to their fundamental nature.

Acknowledgments I thank Jenny Wang and Young-Kwang Nam for collaboration on the schema matching tool, and Kai Lin and Vitaliy Zavesov for work on its implementation. Thanks also to an anonymous referee for many valuable comments, and to Profs. Deutsch, Gottlob, Leitsch, Lenzerini, Papakonstantinou, Rahm, and Vianu for encouragement and/or hospitality; any remaining errors are of course my own responsibility, and are perhaps inevitable in trying to algebraicize a large subarea of computer science in a short time. This research was partially supported by the National Science Foundation under Grant No. ITR 0225676, Science Environment for Ecological Knowledge (SEEK).

2 Background

Information comes from and goes to human beings: pixels, bits, marks on paper, etc. have no meaning in themselves, but must be interpreted. So what is stored in databases (or books, cave walls, memory chips, video tapes, or other media) is not information, but *data*. Interpretation has long been studied by numerous disciplines, including semiotics, hermeneutics, pattern recognition, literary criticism, ethnomethodology, statistics, media studies, psychology, machine learning, phenomenology, cognitive neuro-science, and psychophysics, to name a few. But it is still poorly understood, in part because the narrow confines of individual disciplines prevent a comprehensive perspective on a complex phenomenon that transcends such boundaries. It is distressingly common to sweep as much complexity as possible under the rug of "context," and it is a sadly pervasive error to think that all the contextual information needed to interpret data can be digitally encoded and mechanically applied. Human beings are the ground for all information and all interpretation, and human society is the matrix within which all meaning is embedded; a prevasive result of this situation is the creation of a constantly shifting foreground and background, with the latter being called "context" [12]. I do not say that machines could never deal with this, but they are far from being able to do so today, and to do so, they would have to be given not merely more computing power, more memory, and more sensory capability, but would also have to be embedded in human society, a situation having consequences which not everyone welcomes, although it is beginning to appear in ubiquitous computing, emotive computing, etc.

Confining attention to digital data, and moving up from bits, we find a great variety of storage media, including tape, CD, DVD, flash memory, hard and scuzzy disk, RAM, stick, jukebox, and more. This level is usually taken for granted, but it is non-trivial, as anyone who has ever had to deal with the internals of a device driver can attest. Data at this level is structured into bits, bytes, tracks, and other device-dependent subdivisions, which are oriented towards particular patterns of use.

Databases further organize storage media to facilitate operations that interrogate and update content. This may involve encodings for data types, as well as further structuring, to make relationships between different data items explicit. Types determine what operations are possible on the underlying bits, and provide humans with valuable clues about interpretation, while structuring makes it easier to find related data, (usually) provides associated names, and again facilitates interpretation. Data about data is called **meta-data**. The most important, or at least best understood, meta-data for databases are **schemas**, which describe conventions for structuring, typing, and naming data. However, several different kinds of schema are in common use; when schemas are called "models," kinds of schemas may be called "meta-models," but this paper prefers the term **schema species**. The best established and most studied species is the **relational**, which structures data into sets of relations with fields, while another, called **object oriented**, structures data into objects with inheritance and attributes. With the rise of the web, XML is poised to overtake these; its approach is called **semi-structured**, hinting at its greater flexibility. There are legacy databases that use so called hierarchical meta-models; spreadsheets and formatted ascii files can also be considered to be schema species.

Unfortunately, a great deal of critically important meta-data falls outside the scope of schemas. For example, while it may be possible to say that a certain item of data is measured in feet, a schema cannot say what a "foot" actually is, or relate it to other units, such as meters. In some cases, the way a measurement is taken, called its protocol, can be very important. For example, in ecology, species density is defined as species count divided by area. But for marine species, volume may be the relevant denominator, and it will matter how species counts are obtained, e.g., by a net that is dragged for a certain amount of time, or by observation from some fixed point. The time of year and time of day may also be important, since species migrate at different times of the day and year, and of course variations in weather modify these patterns. The taxonomies used to classify species may also differ, e.g., different criteria may be used, different granularities of classes, etc. Different modes of observation may also have different inherent inaccuracies, some of which may be systematic, e.g., because certain colors are distorted or difficult to distinguish underwater. And all this is just one tip of an enormous iceberg of potentially critical knowledge relevant to the interpretation of data.

As the above discussion suggests, making use of data in one database may require integrating it with data from others. For example, to interpret a measurement of the density of gray whales at a certain time and place, we may need to know the ocean temperature, and the path and time of migration for that species, and then we may need to compare the current data with data from previous years, among other things. This requires knowing what factors are relevant, how important they are, where to get the necessary data, and how to process it. In general, the necessary data is stored in different databases at different sites, and although it might all be accessible over the internet, some of it may require specialized background knowledge. Typically today, scientists import all the data they need into their own lab, massage it in various ways (such as averaging or interpolating time series to achieve compatibility with other time series), and finally process the integrated data, often using hand coded *ad hoc* programs. All this is a far cry from writing a query in a standard database query language such as SQL or XQuery, and then running it over a single database. However, the goal of much current research is to bring data integration closer to that paradigm; some of this research is described in the next three subsections. It is not claimed that these three areas include everything that is relevant, or that they are likely to be sufficient for solving all the problems of data integration and translation.

2.1 Schema Integration

A standard approach to data integration is to construct a global database connected by views to the various local databases such that the global database contains all the data of interest from the local databases, and can be queried to obtain exactly what is needed for some particular purpose. However, it is likely that the local databases are evolving, and it is therefore wasteful to continually update and duplicate everything. For this reason, the global database is often *virtual*, represented by a schema. More formally and generally, using some language from category theory, correct views are schema morphisms, and the situation described above is a cone (and/or co-cone) in the category of schemas, with apex a global schema G over local schema L_i. The case of a cone $v_i \colon G \to L_i$ is called "local-as-view," while the case of a co-cone $u_i \colon L_i \to G$ is called "global-as-view." Cones correspond to relations, and co-cones to co-relations.

Unfortunately, it can take a lot of effort to construct the necessary views, effort that is often not worthwhile for any single project. Moreover, the schemas of the local databases are in general also evolving, along with the needs of the domain of activity involved, so that additional ongoing work is needed to "maintain" the views, i.e., to keep them up to date; often, no single research project has the resources or motivation to do this additional work. Therefore tools are being built to construct schema mappings; although there has been a good deal of effort, one outcome is that total automation is infeasible, so that some human intervention is needed to achieve high quality results; see Section 3.7.

A different kind of gap is that the local databases often use different species of schema, whereas the concept of view is limited to schemas of the same species. An expensive option is to "wrap" a database of one species with a "mediator" providing data over a schema of a different species; this may be practical for legacy databases using obsolete data models, but otherwise may not be worth the trouble. Our solution is to define morphisms between schemas of different species. To avoid an *ad hoc* approach in which there are n^2 different notions of view among n different schema species, it would make sense to have a single general notion of schema that can be used for databases of any species; then we only need one general notion of schema morphism, which should specialize to views for the individual species. Such a notion is described in Section 3; it goes well beyond matching by supporting complex morphisms with semantic functions and conditions, and as far as we know, is the first theory of views that applies to arbitrary species. In addition, techniques in Section 4.2 yield schema "heteromorphisms" which support translation and integration of data over schemas of different species.

The research that seems most closely related to this paper is due to Alagic and Bernstein in [1], which can be interpreted as axiomatizing the schemas, constraints, morphisms, and databases of a fixed species as an institution in which schemas are signatures, constants are sentences, morphisms are theory morphisms, and databases are models. This approach, called the "schema transformation framework," is illustrated with an object oriented species based on Java interfaces with Horn clause logic constraints. The paper also suggests that schema integration is given by pushouts in the category of theories. Further motivation for the use of category theory is given on Bernstein's model management webpage [6].

2.2 Ontologies and Institutions

Computer-based ontologies express logical relations among entities and predicates. The intention is to establish a standard terminology for some body of data, to simplify search and integration. Although many ontology languages are equivalent to Horn clause logic, there are also many other formalisms, some of which even use second order logic. Moreover, not all ontology languages have a formal semantics. Problems with ontologies are similar to those with schema species discussed in Section 2.1, as is our approach to solving them, by providing a uniform formalization of logics and their morphisms. While logicians have been reluctant to formalize the notion of "a logic," computer scientists have been less shy, and there is now a considerable literature on *institutions* [20, 35], which formalize logics as an abstraction of satisfaction relations between sentences and models, in the style of Tarskian semantics, but parameterized over a category of signatures (signatures typically supply non-logical symbols, such as predicate and function symbols, for constructing sentences and models). Institutions can be considered a theory of truth, i.e., of sound reasoning, that is relativized to context. A very high level of generality is achieved through the use of category theory. However, it must be noted that the problems raised by social aspects of interpretation become even more acute at the ontology level than at the schema and data levels.

2.3 Workflows

The kinds of data processing required for applications to scientific research and e-commerce go well beyond what can be accomplished using only database query languages. Data must flow through a complex pipeline, being massaged and combined with other data in a great variety of ways; such processes are called *workflows*. In the case of scientific workflows, more than just data translation or even complex data massaging is needed, including the use of powerful statistics packages, integration with complex scientific models, and display of selected data using visualization packages; there may also be feedback loops, as hypotheses, algorithms and data are progressively refined. Web-based business workflows have similar complexity, although the components are different. A number of languages have been developed to describe workflow components and processes, and of course there are also many *ad hoc* solutions. In general, it is a major task to construct a workflow for a specific project using these technologies, and therefore it is a major open problem to develop more flexible and user-friendly approaches.

Service integration is a generalization of data integration, and the problems described above also arise in this new and more complex setting. We expect that many aspects of our proposed solutions for data integration will extend to workflow and service integration. In particular, our schemas can describe the semantics of ports (i.e., interfaces) of actors in a workflow, and schema morphisms can provide sound translations among ports that have different requirements, that are expressed in different formalisms.

3 Schemas and their Morphisms

Subsections below review some algebra that we need, and then develop abstract schemas, their databases, queries, constraints, and morphisms. Section 3.6 assumes some category theory. A final subsection briefly describes a tool that implements this theory for some special cases.

3.1 Brief Introduction to Algebraic Specification

This subsection briefly reviews algebraic specification [22, 26, 27]. A (**many sorted algebraic**) **signature** Σ is a **sort set** S and a set (also denoted Σ) of operation symbols, each with a **rank**, which is a string of input sorts and a single output sort; a **constant** of sort s has rank $([], s)$, where $[]$ is the empty string. Elements of S are also called **types**. A Σ**-algebra** A is a set A_s of elements of sort s for each $s \in S$, plus a function $A_\sigma : A_{s1} \times ...A_{sn} \to A_s$ for each operation symbol σ of rank $(s1...sn, s)$; A_s is the **carrier** of sort s. **Unsorted** signature and algebra are the same as one sorted, but instead of rank, the **arity** of an operation is its number of arguments (see also Example 20). In **order sorted algebra**, the sort set S is partially ordered by a **subsort** relation, and if s is a subsort of s', then $A_s \subseteq A_{s'}$ and overloaded operations are consistent with all such inclusions (see [22] for details).

A Σ**-term** is a well formed expression (with respect to sorts, subsorts, and arties) built using function symbols from Σ plus possibly some variable symbols, and a Σ**-equation** is a pair of such terms, considered as universally quantified over variable symbols. A Σ-algebra A **satisfies** a Σ-equation iff all substitution instances of the two terms, using values in A for the variable symbols, are equal in A. A **conditional** Σ-equation has an additional set of pairs of terms, and is **satisfied** by A iff the first two terms are equal in A under a substitution whenever all the other pairs of terms are equal under the same substitution. We can write $A \models_\Sigma e$ when A satisfies e.

An **algebraic specification** is a pair (Σ, E) of a signature Σ and a set E of equations using only operations in Σ. An algebra **satisfies** (Σ, E) if it has signature Σ and satisfies all equations in E, some of which may be **conditional**, i.e., required to hold only when a given unconditional equation, called its condition, already holds. Equations are (usually implicitly) universally quantified over their variables, each of which must have a declared sort. The **term algebra** T_Σ of a signature Σ consists of all well-formed terms built from the operations (including constants) in Σ. The **initial algebra** of a specification (Σ, E) is T_Σ with identifications determined by E; formally, it is the quotient T_Σ/E of T_Σ by the congruence generated by the ground instances of equations in E; it satisfies the specification by construction. Term algebras are initial for the empty set of equations. The term "initial" is used because there is a unique homomorphism from it to any other algebra satisfying the specification. Any two initial algebras for a given specification are isomorphic, i.e., they are the same up to a renaming of their elements.

We also need parameterized algebraic specifications, called **polymorphic types** or **polytypes**, such as sets, bags, lists, and pairs; more exotic types, such as graphs, tables, and balanced binary trees, are also possible. For the purposes of this paper, a polytype is an algebraic specification with one or more type parameter, for which we will later substitute previously defined sorts[1].

Example 1. Here two simple examples of polytypes, written in the notation of OBJ [29, 22], in which the keyword "`obj`" indicates initial semantics:

```
obj Pair[X Y :: TRIV] is sort Pair .
```

[1] A more general definition (e.g., [20, 25, 28]) has "interface theories" that specify multiple sorts and operations to be substituted, subject to axioms that restrict the allowable substitutions.

```
    op [_,_] : X Y -> Pair .
    op 1_ Pair -> X .
    op 2_ Pair -> Y .
    var X : X .  var Y : Y .
    eq 1 [X, Y] = X .
    eq 2 [X, Y] = Y .
  endo

  obj ListOf[X :: TRIV] is sort List NeList .
    subsort X < NeList < List .
    op nil : -> List .
    op cons : X List -> NeList .
    op head_ : NeList -> X .
    op tail_ : NeList -> List .
    var X : X .  var L : List .  var N : NeList .
    eq head(cons(X, L)) = X .
    eq tail(cons(X, L)) = L .
  endo
```

The keywords `obj` and `endo` mark the beginning and end of a specification, and the string following `obj` is its name, while the material between `[` and `]` declares the type variables; `TRIV` indicates that there are no restrictions on these variables. The first sort declared is called the principal sort of the specification. Underbars in operation declarations indicate where arguments go in mixfix syntax, and the rank follows the colon. Of course, there are also many other ways to specify lists, such as the following:

```
  obj ListOf[X :: TRIV] is sort List NeList .
    subsort X < NeList < List .
    op nil : -> List .
    op __ : NeList List -> NeList [assoc id: nil].
    op __ : List NeList -> NeList [assoc id: nil].
    op __ : NeList NeList -> NeList [assoc id: nil].
    op head_ : NeList -> X .
    op tail_ : NeList -> List .
    var X : X .  var L : List .  var N : NeList .
    eq head(cons(X, L)) = X .
    eq tail(cons(X, L)) = L .
  endo
```

where "`[assoc id: nil]`" indicates that the associative and identity laws hold for the append operation (with juxtaposition syntax indicated by `__`), with `nil` its identity constant. □

The **constructors** of a specification are those the target sort of which is the principal sort of the specification, while its **selectors** are the inverses of constructors, as defined by equations. Thus, `[_,_]` is a constructor, while `1_` and `2_` are selectors in `PAIR`; the type parameters for `Pair` are `X` and `Y`. There may also be some predicates, given as Boolean valued functions (the sort `Bool` for Booleans is imported into every specification). To **instantiate** a parameterized specification, we simply substitute names of types for the type variables in its name. For example, `ListOf[Nat]` gives lists of natural numbers, and `Pair[Bool,Bool]` gives pairs of Booleans; all the equations should also be instantiated by substitution of the actual type for the generic type variable.

It is also natural to write `ListOf(Nat)`, `Pair(Bool, Bool)` (but `Nat`$\times$`Bool` is even better), `ListOf(Pair(Nat, Bool))`, etc. for the principal sorts that results from instantiation, and this notation is used in the type algebras constructed in Sections 3.2 and 3.3.

A **collector** is a polytype with constructors for inserting elements of its parameter type, and a Boolean valued operation `_in_`, also written `_`$\in$`_`, for querying whether an element has been inserted, but without operations for removing arbitrary inserted elements. Thus, `SetOf`, `BagOf`, `ListOf` are collectors, but pairs and arrays are not. In several places, we need to *iterate* a binary operation over one collector structure to build a value, or to build another collector structure, i.e., to apply the operation iteratively to the elements of the given structure. For example, if L has type `ListOf[String]` and if P is a Boolean valued function on `String`, then the result of iterating the operation $P(x) \wedge B$ over L may be written $\bigwedge_{x\in L} P(x)$. Or if L is a list of pairs of integers, then the result of iterating $\texttt{insert}(p_2(x), S)$ is $\{p_2(x) \mid x \in L\} = \{b \mid \langle a, b\rangle \in L\}$, where `insert` is the set collector and $p_2(x)$ extracts the second component of x. Such polymorphic **iterator** operations are are similar to `mapcar` in Lisp, and are easily defined by recursive equations over basic collector constructors (e.g., see [15]).

Values that depend on **conditions** can be constructed using the polymorphic `if_then_else_` function, the first argument of which is Boolean while the other two are of some other type, such as integer, as in the expression $\texttt{if}\ \mathit{even}(n)\ \texttt{then}\ n+1\ \texttt{else}\ n+2$.

Example 2. What are called **null values** in databases can be handled in a simple and elegant way in order sorted algebra. For example, if `Int` is the sort of integers, then we can a supersort `Int?` of `Int`, and a new constant `nullInt` of sort `Int?`. Formally, in OBJ syntax, we can write

```
obj NULLINT is pr INT .
  sort Int? .  subsort Int < Int? .
  op nullInt : -> Int? .
endo
```

Here "`pr INT`" indicates that the already defined module `INT` for integers is imported. It is clear that null values can be defined in the same way for Booleans, characters, or any other sort of data value. □

To ensure uniform representation for interoperability in our database applications, it is convenient to assume that the basic values used for data are taken from a fixed algebra D, called the **data algebra**. We will assume that all the operations we need on data values are included in the signature Σ^D of

D, and we also assume for convenience that all elements of D are included in Σ^D as constants. Let S^D denote the set of sorts of D, elements of which may be called **basic types**; typical basic types are `Int`, `Bool`, `Char`, and `String` (of characters), and typical operations are addition, exclusive-or, append, etc.

We can view any data algebra as an initial algebra of a suitable specification, as follows: let E^D be the set of all equations that are satisfied by D; then D is Σ-isomorphic to T_{Σ^D}/E^D. Our formalization of schema morphisms will use a related term algebra, based on the following construction: if X is an S^D-sorted signature containing only constants (to be regarded as new variable symbols), let $\Sigma(X)$ denote the union signature $\Sigma \cup X$, and let $T^D(X) = T_{\Sigma^D(X)}/E^D$; as usual, we denote elements of $T^D(X)$ by elements of $T_{\Sigma^D(X)}$, such as $x + 2y$ or `if` $even(n)$ `then` $n + 1$ `else` $n + 2$; note that in $T^D(X)$, the latter is *actually equal* to $2\lceil\frac{x}{2}\rceil + 1$.

A different approach to null values than that of Example 2, called **labelled nulls** in the database literature, adds new function symbols of sort s as null values of sort s, for example, to handle key constraints in schema based data translation, e.g., in [32, 38]; they may be just constants, but can also have non-trivial arity, in which case they are Skolem null values; intuitively, they represent values that are presently unknown, but that depend on other values. Because T^D can be extended to $T^D(\Omega)$ where Ω contains nulls, we can assume that T^D contains whatever nulls are needed. Note that labelled null operations can also be put in error supersorts, as in Example 2.

3.2 Schemas

Schemas may contain both structure declarations and semantic constraints, which describe those aspects of a database that should be invariant under transactions; a particular schema of a given species describes the databases that conform to that schema. This subsection presents general notions of schema and species, with some examples.

The first ingredient of a schema species is a data algebra D, consisting of the basic data elements and operations upon them, having one sort for each type of basic element. Elements of D serve as data values in schema instances, i.e., databases or "e-documents." Different schema species may have different data algebras, depending on the constants, functions and relations that they need.

The second ingredient of a schema species is a collection $\mathcal{P}$ of polytypes that includes the finite set polytype. From this and a data algebra D, we will build the *type algebra*, which plays a fundamental role in our theory. The following are needed for its construction: For each constructor P_i in $\mathcal{P}$, let c_i be an operation on types (not on data) having the same (unsorted) arity as P_i; examples are unary `SetOf` and `ListOf`, and binary `Pair`, the latter of which is hereafter written as an infix product $\times$. The constructors in P_i structure data, for example, a relation as a set of records, whereas the c_i are constructors for type expressions, which describe the structure of data, as in the term `SetOf(StudentID` $\times$ `CourseID)`. Let $C_{\mathcal{P}}$ or just C denote the (unsorted) signature containing all c_i with their arities, and all sorts S^D of the data algebra D as constants. We will assume that all the iterators we need are already in $\mathcal{P}$.

Type constructors with just one type argument are often called **collectors**; examples are `SetOF`, `List Of`, and `BagOf`, as well as more exotic examples, such as `2Arr`(N, K), for 2-dimensional $(N \times K)$-arrays, e.g., `2Arr(366,24)(Real)`, for recording hourly temperatures over a year (with a null value for the 366th day of a non-leap year).

Definition 1. Given a data algebra D, a collection $\mathcal{P}$ of polymorphic type constructors, and set N of elements called **names** (which will be used to name database (sub)structures) in the form of strings, i.e., elements of $D_{\texttt{String}}$, let the **type algebra**, denoted $T(N)$, consists of all **type expressions**, which are well formed terms built using $C_{\mathcal{P}}$ and N. □

Note that $T(N)$ has just one sort. Type expressions are fundamental for our abstract schemas (Definition 3 below). It is not as well known as it should be that type expressions form an initial algebra, nor that abstract syntax and Knuthian attribute grammars are also elegantly handled by initiality [26, 27].

The third ingredient of a schema species is restriction on the allowable forms for schemas of that species, given as an admissibility relation $\mathcal{R}(n, t)$ between names n and type expressions t, and generally defined by a set of rules. We therefore have the following:

Definition 2. A **schema species** consists of a data algebra D, a set $\mathcal{P}$ of polymorphic types, and an **admissibability relation** $\mathcal{R}$ between names and type expressions. □

Example 3. **Relational Database Species.** Let P_1 define finite sets, and let P_k for $k > 1$ define k-tuples, so that $c_1 = \texttt{SetOf}$ and c_2 is the binary infix product type constructor $\times$, c_3 is a constructor for types of 3-tuples, etc[2]. Then the following defines admissibility for relational schemas, where the database name is R, the names for its relations are $R_1, ..., R_n$, and the names for the fields of each R_i are F_{ij} for $i = 1, ..., n$ and $j = 1, ..., K_i$:

$$\mathcal{R}(R, R_1 \times ... \times R_n)$$
$$\mathcal{R}(R_i, \texttt{SetOf}(F_{i1} \times ... \times F_{iK_i}))$$
$$\mathcal{R}(F_{ij}, d) \text{ for } d \in S^D$$

The first says a relational database has relations $R_1, ..., R_n$ for some $n > 1$, the second says each R_i is a set of records of type $F_{i1} \times ... \times F_{iK_i}$ having $K_i > 0$ fields F_{ij} which the third says all have values of a basic type. (If preferred, `BagOf` could be used instead of `SetOf`.) Any particular relational schema makes particular choices for these parameters, as illustrated below. □

The **name graph** $G(\mathcal{S})$ of a function $\mathcal{S}\colon N \to T(N)$ with a given top name Top has Top as its root node, and if t is a node of $G(\mathcal{S})$ and if a name n appears in $\mathcal{S}(t)$, then there is an edge from t to n in $G(\mathcal{S})$. $\mathcal{S}$ is **acyclic** if its graph $G(\mathcal{S})$ is acyclic. Also, name $n \in N$ is **reachable** if there is a path in $G(\mathcal{S})$ from Top to n, and $\mathcal{S}$ is **reachable** if every name in N is reachable. The following captures the structural aspect of schemas (constraints are considered later):

[2] A more sophisticated approach uses only the binary pair constructor with an associative law, so that n-tuples are built using $n - 1$ pair constructors.

Definition 3. An **abstract schema** of a given species is a finite set N of elements called **names** containing a **top name** designated Top, plus a function $\mathcal{S} : N \to T(N)$ such that every term $\mathcal{S}(n)$ is admissible for n, and has depth 1 or 2 (where constants have depth 1), and such that any term obtained by starting from some $\mathcal{S}(n)$ and making any number of substitutions of $\mathcal{S}(n')$ for occurrences of names n' in the resulting terms are also admissible for n. A schema is **acyclic** if its name graph is acyclic, and is **reachable** if its name graph is reachable with respect to its top name. Let N_* denote the set of reachable names of $\mathcal{S}$. $\Box$
Abstract schemas capture the abstract structure of concrete schemas, including their relations of subordination, ordering, and typing. Restricting the terms $\mathcal{S}(n)$ to depth 1 or 2 guarantees that names are associated with substructures, which enhances readability, and prevents *anonymous types*, which are subexpressions without assigned names.

Example 4. **A Relational Database Schema.** Here is a schema for a simple relational student database having three relations:

$\mathcal{S}(\texttt{SDB}) = \texttt{Student} \times \texttt{Enrolled} \times \texttt{Course}$
$\mathcal{S}(\texttt{Student}) = \texttt{SetOf}(\texttt{StudentID} \times \texttt{Address} \times \texttt{Major} \times \texttt{GPA})$
$\mathcal{S}(\texttt{Enrolled}) = \texttt{SetOf}(\texttt{StudentID} \times \texttt{CourseId})$
$\mathcal{S}(\texttt{Course}) = \texttt{SetOf}(\texttt{CourseId} \times \texttt{Synopsis})$
$\mathcal{S}(\texttt{StudentID}) = \texttt{Nat}$
$\mathcal{S}(\texttt{Address}) = \texttt{String}$
$\mathcal{S}(\texttt{Major}) = \texttt{String}$
$\mathcal{S}(\texttt{GAP}) = \texttt{Real}$
$\mathcal{S}(\texttt{CourseID}) = \texttt{Nat}$
$\mathcal{S}(\texttt{Synopsis}) = \texttt{String}$

where `Nat` is for natural numbers. The type algebra contains terms such as $\texttt{SetOf}(\texttt{CourseId} \times \texttt{Synopsis})$, and $\texttt{Student} \times \texttt{Enrolled} \times \texttt{Course}$, which uses the 3-tuple type constructor, plus infinitely many others not used by $\mathcal{S}$. $\Box$

Claims of Siméon and Wadler [40] about the "essence of XML" can be seen as implying that the abstract schemas of XML Schemas use only the collector `ListOf`, reflecting the inherent ordering in XML syntax; this is of course a simplification, since the official specification of XML is very large (over 300 printed pages) and ugly. The following captures this essential abstract structure, including relations of subordination, ordering, and typing; note that elements and attributes are not distinguished.
Definition 4. The **essential XML species** uses only the list and k-tuple (for $k > 1$) polytypes; any term involving only these is admissible for any name. $\Box$
Most of our examples are of this species.
Example 5. We illustrate this with a schema for lists of persons:

$\mathcal{S}(\texttt{PDB}) = \texttt{ListOf}(\texttt{Person})$
$\mathcal{S}(\texttt{Person}) = \texttt{Name} \times \texttt{Age} \times \texttt{Mother} \times \texttt{Father}$
$\mathcal{S}(\texttt{Name}) = \texttt{String}$
$\mathcal{S}(\texttt{Age}) = \texttt{Nat}$
$\mathcal{S}(\texttt{Mother}) = \texttt{Person}$
$\mathcal{S}(\texttt{Father}) = \texttt{Person}$

The name graph of this schema is cyclic, because the type name `Person` occurs within its own definition. In this example, because lists are finite, null values would be used to avoid an infinite regress of ancestors; theoretically this schema allows circularities in its databases, e.g., a person could be its own father and mother, although a constraint could eliminate this possibility. □

Our second illustration of Definition 3 is a simple acyclic schema for lists of books; it also includes an XML Schema for the same structure:

Example 6. **Abstract and Concrete XML Schemas for Books.** Here the top name is `Bib`, and

$$\mathcal{S}(\texttt{Bib}) = \texttt{ListOf}(\texttt{Book})$$
$$\mathcal{S}(\texttt{Book}) = \texttt{Title} \times \texttt{Year} \times \texttt{Author}$$
$$\mathcal{S}(\texttt{Author}) = \texttt{ListOf}(\texttt{author})$$
$$\mathcal{S}(\texttt{Title}) = \texttt{String}$$
$$\mathcal{S}(\texttt{Year}) = \texttt{Int}$$
$$\mathcal{S}(\texttt{author}) = \texttt{String}$$

The XML Schema for this (taken from [13]) looks much more complex:

```
<xsd:group name="Bib">
  <xsd:element name="bib">
    <xsd:complexType>
      <xsd:group ref="Book" minOccurs="0" maxOccurs="unbounded"/>
     </xsd:complexType>
   </xsd:element>
</xsd:group>

<xsd:group name="Book">
  <xsd:element name="book">
    <xsd:complexType>
      <xsd:element name="title" type="xsd:string"/>
      <xsd:element name="year" type="xsd:integer"/>
      <xsd:element name="author" type="xsd:string"
           minOccurs="1" maxOccurs="unbounded"/>
    </xsd:complexType>
  </xsd:element>
</xsd:group>
```

To explain the details of this XML Schema in words would take much space, but note that the names in abstract schema are just the strings following "`name=`" above, except that collectors do not need names, and `bib` and `book` are not needed. Our abstract schema $\mathcal{S}$ does not express the constraints on the number of items in the two lists. If the list polytype includes the empty list, then the constraint that `Author` lists are non-empty needs a constraint, for which see Example 14 and Definition 10. □

3.3 Explicit Type Algebra

Database entities often have the same underlying type, such as integer, but use it in different ways. To distinguish them, we "wrap" their underlying type names with explicit names that indicate how they are used; we call this an *explicit type*, and use a quotient construction to identify the explicit type with the name given in its schema.

Definition 5. Given a schema species and a schema $\mathcal{S}\colon\ N \to T(N)$ of that species, let $T^{\#\#}(N)$ denote the free unsorted algebra consisting of all well formed terms built using N and S^D plus the set of all unary type constructors $\texttt{\#}n$ for $n \in N$ plus the (unsorted) type constructors in $C_{\mathcal{P}}$ for that species. Elements of $T^{\#\#}(N)$ are called **explicit type expressions**, and $T^{\#\#}(N)$ is called the **explicit type algebra**. A **type equation** of $\mathcal{S}$ has the form $n = \texttt{\#}n(\mathcal{S}(n))$ for $n \in N$; let E be the set of these. Then $T^{\#\#}(N)/E$ is called the **reduced explicit type algebra**, and is denoted $T^{\#}(N)$.

Each E-equivalence class of type expressions contains a unique **complete** explicit type expression, the result of substituting $\texttt{\#}n(\mathcal{S}(n))$ for n wherever possible, except that no name is expanded twice along any path, i.e., such that no operation $\texttt{\#}n$ appears more than once along any path in the expression. A **necessary** type expression is a complete subexpression of some explicit type expression equal under E to the top sort of the schema. □

Expressions in $T^{\#\#}(N)$ extend those in $T(N)$ by attaching names to subexpressions using `#` operations. For example, `#Title(String)` and `#author(String)` are different in $T^{\#\#}(N)$ but would be just `String` in $T(N)$; also, paths can be traced by following `#` operations in the parse trees of explicit type expressions. The reduced explicit type algebra identifies names with the explicit type expressions that they denote under $\mathcal{S}$. Note that E-classes can be infinite for cyclic schemas, e.g., for the schema `PDB` of Example 5.

Example 7. **Complete Type Expressions.** The complete type expression for the top sort of the student database of Example 4 is

```
#SDB(SetOf(#Student(#StudentID(Int) × #Address(String) × #GPA(Real))
    × SetOf(#Enrolled(#StudentID(Int) × #CourseId(Nat)))
    × SetOf(#Course(#CourseId(String) × #Synopsis(String))))
```

It is interesting to look at the complete type expression of the schema in Example 5, because it involves recursion. The complete top type expression is

```
#PDB(ListOf(#Person(#Name(String) × #Age(Nat)
    × #Mother(Person) × #Father(Person))) ,
```

in which the second two `Person` names are *not* expanded. □

Example 8. **Bibliographic Schema Type Expressions.** Some complete type expressions for the schema of Example 6 are

```
ListOf(#Book(Title × Year × Author))
```

and

$\texttt{ListOf(Title} \times \texttt{Year} \times \texttt{ListOf(author))}$

so that modulo the type equations, `Bib` is actually equal to `#Bib(ListOf(Book))` as well as to the following complete type expression for the schema,

$$\texttt{\#Bib(ListOf(\#Book(\#Title(String)} \times \texttt{\#Year(Int)} \times \\ \texttt{ListOf(\#Author(ListOf(\#author(String)))))))} \;.$$

Unused names like `GenomeRef` will not appear among the necessary type expressions of $\mathcal{S}$. □

Example 9. A schema can be recovered from its complete type expression by a process that we call **unfolding**, which successively removes each outermost `#`n operation from the term, and then defines $\mathcal{S}(n)$ to be the remainder of the term. Rather than give a formal definition, we do a simple example, unfolding the complete type expression given in the above example for the schema in Example 6. The first step produces the equation

$$\begin{aligned}\mathcal{S}(\texttt{Bib}) = (&\texttt{ListOf(\#Book(\#Title(String)} \times \texttt{\#Year(Int)} \times \\ &\texttt{ListOf(\#Author(ListOf(\#author(String)))))))} \;,\end{aligned}$$

and the second step produces the equations

$$\begin{aligned}\mathcal{S}(\texttt{Bib}) &= \texttt{ListOf(Book)} \\ \mathcal{S}(\texttt{Book}) &= \texttt{\#Title(String)} \times \texttt{\#Year(Int)} \times \\ &\quad\texttt{ListOf(\#Author(ListOf(\#author(String))))} \;,\end{aligned}$$

while the third produces

$$\begin{aligned}\mathcal{S}(\texttt{Bib}) &= \texttt{ListOf(Book)} \\ \mathcal{S}(\texttt{Book}) &= \texttt{Title} \times \texttt{Year} \times \texttt{ListOf(Author)} \\ \mathcal{S}(\texttt{Title}) &= \texttt{String} \\ \mathcal{S}(\texttt{Year}) &= \texttt{Int} \\ \mathcal{S}(\texttt{\#Author}) &= \texttt{(ListOf(\#author(String))))} \;,\end{aligned}$$

which after one more step yields the original bibliographic schema. □

3.4 Specification and Databases of a Schema

The following is the basis for defining the databases of a schema:

Definition 6. The **specification** $\mathit{Spec}(\mathcal{S})$ of a schema $\mathcal{S}\colon\, N \to T(N)$ has:

1. as its sorts, the necessary type expressions of $\mathcal{S}$;
2. as its operations, those of the instantiated polytypes that appear in some necessary type expression of $\mathcal{S}$;
3. as its equations, those of the instantiated polytypes that appear in some necessary type expression of $\mathcal{S}$, plus E^D.

The **signature** of $\mathcal{S}$ consists of the first two items, and the top name of $\mathcal{S}$ is called the **top sort** of its specification. □

Thus the sorts of $\mathit{Spec}(\mathcal{S})$ are the type expressions that are actually used in $\mathcal{S}$.

Example 10. **Bibliographic Schema Specification.** For $\mathcal{S}$ the schema of Example 6, the constructors in its specification include the following:

$cons$: `Book Bib` $\rightarrow$ `Bib`
$cons$: `author Author` $\rightarrow$ `Author`
$c3$: `Title Year Author` $\rightarrow$ `Book`

where $c3$ constructs 3-tuples (plus a couple of empty list constants). The equations include $head(cons(a, L)) = a$ for a a variable of sort `Book` and L of sort `Bib`; another such equation is $tail(cons(a, L)) = L$. □

There is a confusion about the word "model": some authors use it for schemas (e.g., the "model management" of [5]), while others use it for the databases that conform to a schema, by analogy with use of this word in model theory (in logic) for a structure that satisfies a theory. This analogy is better than the schema analogy, but is still not exact, as the following shows:

Definition 7. Given a schema species and a schema $\mathcal{S}\colon N \rightarrow T(N)$ of that species, then a **database implementation of** $\mathcal{S}$ is an initial algebra I of the specification of $\mathcal{S}$ such that its restriction[3] to Σ^D is D; we will write $I_{\mathcal{S}}$ for such an I. A **database state** of I is an element B of the carrier of the top sort of I, and we also say that B is a **conforming database** for $\mathcal{S}$ and write $B \models \mathcal{S}$. □

While abstract data type theory [27] abstracts away from how the elements of an initial algebra are represented, for a schema $\mathcal{S}$, we are interested in different concrete initial algebras, since they correspond to different implementations of databases of $\mathcal{S}$, with the elements of each such algebra being the database states in that representation, where data sorted subterms are given by constants from D. The following illustrates these ideas:

Example 11. **A Student Database.** Among the database implementations of the student schema in Example 4 is an initial algebra that uses conventional notation for the tuples resulting from product constructors and for the sets resulting from set constructors. In this algebra, a typical small student database looks as follows:

$$\begin{array}{l} \langle\, \{\, \langle 215, \textit{Muir}\, 129, \textit{Math}, 3.2\rangle\,,\ \langle 329, \textit{Revelle}\, 774, \textit{CS}, 3.8\rangle\, \}\,, \\ \quad \{\, \langle 215, 130\rangle, \langle 215, 220\rangle, \langle 329, 130\rangle, \langle 329, 87\rangle\, \}\,, \\ \quad \{\, \langle 130, ...\rangle,\ \langle 87, ...\rangle,\ \langle 220, ...\rangle\, \}\, \rangle \end{array}$$

where ... indicates omitted text that describes courses. □

Constraints and schema morphisms need notation for selectors. If there is a standard notation, we can use it, e.g., `head` and `tail` for lists; selectors for the product type constructor $\times$ will be denoted by p_k where the desired argument is in the k^{th} place, or by their argument type names prefixed by `&` (e.g., two selectors from Example 4 are `&Enrolled` and `&GPA`). The following is also needed:

Definition 8. Given a schema species, a schema $\mathcal{S}\colon N \rightarrow T(N)$ of that species, and reachable $n \in N$, then the **set extractor** from Top to n, denoted `##`n, is defined by the formula below, in which B is a database conforming to $\mathcal{S}$ and $p = s_1...s_k$ the sequence of selectors (no collectors) on a path from Top to n, in the parse tree t of the complete type expression for $\mathcal{S}$,

[3] Because D is also an initial algebra, this does not compromise the initiality of I.

$$\texttt{\#\#}n(B,p) = \{e_k \mid e_k \texttt{ in } s_k(e_{k-1}), ..., e_2 \texttt{ in } s_1(B)\} \ ,$$

where $e' \texttt{ in } s(e)$ means $e' = e$ when there is no collector directly above s in t, and means $e' \texttt{ in } s(e)$ using `in` of the collector directly above s if there is one. □

Note that $\texttt{\#\#}n$ is an iterator over the element insertion operation for the set polytype. Usually there is a unique path from Top to n, in which case we write just $\texttt{\#\#}n(B)$. The following illustrates how `##` works:

Example 12. For the structure of Example 4, the selectors on the path from `SDB` to `CourseID` are p_3, p_2. Therefore $\texttt{\#\#CourseId}(B) = \{e \mid e = p_2(S), S \in p_3(B)\} = \{p_2(S) \mid S \in p_3(B)\}$, which for B the database of Example 11 has the value $\{130, 87, 220\}$. □

3.5 Queries and Constraints

Definition 9. Given a schema species and a schema $\mathcal{S} \colon N \to T(N)$ of that species, a **query** over $\mathcal{S}$ is a set valued expression with a single variable of the top sort of $\mathcal{S}$, constructed from the sets $\texttt{\#\#}n$ for reachable $n \in N$, plus standard Boolean operations, standard set operations, and operations in Σ^D. □

Set comprehension over Boolean expressions is especially important; because databases in our theory are terms in an initial algebra, these are necessarily finite, and set comprehension is just iteration over single element insertion, which is assumed to be in $\mathcal{P}$.

Example 13. The following is a query over the schema of Example 4:

$$\{p_1(S) \mid S \in \texttt{\&Student}(B) \wedge \texttt{\&GPA}(S) > 3.5\}$$
$$\cap \ \{p_1(E) \mid E \in \texttt{\&Enrolled}(B) \wedge 87 \in p_2(E) \wedge 130 \in p_2(E)\}$$

It asks for the IDs for all students with GPA more than 3.5 who are enrolled in both 87 and 130. □

Although not very user friendly, this notation is convenient for our theory; it is not part of the user interface of our tool discussed in Section 3.7. The database literature distinguishes among a number of different kinds of query; for example, the above is a **conjunctive query**, constructed using conjunction as the only Boolean operation and intersection as the only set operation. Constraints often use Boolean valued iterators, which are abbreviated as "bounded quantifiers," e.g., $(\forall e \in \texttt{\#\#}n(B))\ t$ abbreviates $\bigwedge_{e \in \texttt{\#\#}n(B)} e(t)$, the iteration of conjunction over the elements of $\texttt{\#\#}n(B)$. Similarly, $(\exists e \in \texttt{\#\#}n(B))\ t$ abbreviates $\bigvee_{e \in \texttt{\#\#}n(B)} e(t)$, the iteration of disjunction over the elements of $\texttt{\#\#}n(B)$.

Definition 10. Given a schema species and a schema $\mathcal{S} \colon N \to T(N)$ of that species, a **constraint** for $\mathcal{S}$ is a Boolean term with one free variable B of sort Top, built from set extractors, standard Boolean functions, standard set operations, and operations in Σ^D, usually written $(\forall B \colon Top)\ t(B)$. A database M conforming to $\mathcal{S}$ **satisfies** a constraint $(\forall B \colon Top)\ t(B)$ iff $t(B \leftarrow M) = \texttt{true}$ in $I_{\mathcal{S}}$, where $\leftarrow$ indicates substitution.

A **data integrity constraint** for $\mathcal{S}$ has the form

$(\forall B\colon \mathit{Top})\ (\forall e_1 \in \texttt{\#\#}n_1(B))...(\forall e_k \in \texttt{\#\#}n_k(B))\ t(B)$
where t is a Boolean term built from set extractors and operations in Σ^D.

A **constrained schema** is $(\mathcal{S}, \mathcal{C})$ where $\mathcal{S}$ is a schema and $\mathcal{C}$ is a set of constraints for $\mathcal{S}$. A database **satisfies** $(\mathcal{S}, \mathcal{C})$ if it conforms to $\mathcal{S}$ and satisfies every constraint in $\mathcal{C}$, in which case we write $B \models (\mathcal{S}, \mathcal{C})$. □

Example 14. **Data Integrity Constraints.** The following are typical of data integrity constraints that might be used for Example 6,

$(\forall Y\colon \texttt{Year})\ Y \leq 2005$
$(\forall A\colon \texttt{Author})\ |A| > 0$

where $|\_|$ is a length function assumed to be in the list specification. These do not have the required form, but can be put into it by viewing $\leq$ and $>$ as Boolean valued functions, and using bounded quantifiers:

$(\forall B\colon \mathit{Top})\ (\forall Y \in \texttt{\#\#Year}(B))\ Y \leq 2005$
$(\forall B\colon \mathit{Top})\ (\forall A \in \texttt{\#\#Author}(B))\ |A| > 0$

Constraints that have more than one variable have explicit forms with more than one `##` operation. Letting $\mathcal{C}$ contain the two constraints above, a bibliography database B satisfies $(\mathcal{S}, \mathcal{C})$ iff in fact the year of each book is 2005 or less, and every book has at least one author. □

Example 15. **Equality Constraints.** Another important class of constraints are defined by equations, which can be conditional. For example, in the relational species, what are called **functional constraints** have the form

$(\forall R, R'\colon \texttt{Rel})\ F_1(R) = F_1(R') \Rightarrow F_2(R) = F_2(R')$

where `Rel` names some type of tuple and where F_1, F_2 are fields (i.e., selectors) for that type of tuple. Similarly, a **key constraint** has the form

$(\forall R, R'\colon \texttt{Rel})\ F_1(R) = F_1(R') \Rightarrow R = R'$

in which F_1 functions as a "key" for the type `Rel`.

3.6 Schema Morphisms and Data Translation

In a series of papers (e.g., [5, 1, 7]), Philip Bernstein has presented a vision of "model management" (though it might better be called "schema management"), arguing that schema mappings are the key to many important applications, including generating a website wrapper, populating a data warehouse from data sources, translating relational to XML schemas, integrating a collection of data sources into a single virtual source, and more. Our schema morphisms are more general and more precise than the mappings of Bernstein, but perform the same functions, since they induce translations between databases that conform to the schemas. This section focuses on schema morphisms of a single species, while Section 4.2 extends the theory to "heteromorphisms" which relate schemas of different species.

Before giving the formal definition of schema morphism, we give an informal example to illustrate both **semantic functions**, which manipulate data values (e.g., for combining first and last names to get a full name), and **conditions**, which restrict the application of translation formulae; these are not usually treated in either theory of tools.

Example 16. **An Author Database Translation.** Let the schema $\mathcal{S}_1$ be like $\mathcal{S}$ of Example 6 except for adding the following type definitions,

$$\begin{aligned}
&\mathcal{S}_1(\texttt{author}) = \texttt{fname} \times \texttt{lname}\\
&\mathcal{S}_1(\texttt{fname}) \ = \texttt{NString}\\
&\mathcal{S}_1(\texttt{lname}) \ = \texttt{NString}
\end{aligned}$$

where `NString` is a basic type having no space character, and where `fname` and `lname` are elements for the first and last names of authors. Let $\mathcal{S}_2$ be a second schema built on $\mathcal{S}$ of Example 6 by adding the following type definition and constraint

$$\begin{aligned}
&\mathcal{S}_2(\texttt{author}) = \texttt{ListOf(NString)}\\
&(\forall a\colon \texttt{author})\ 0 < |a| < 3
\end{aligned}$$

Now we can define $\mathcal{M}\colon\ \mathcal{S}_1 \to \mathcal{S}_2$ to map $\mathcal{S}_1$ databases to $\mathcal{S}_2$ databases, using functions that preserve the structure of the components in Example 6, plus

$$\mathcal{M}_{\texttt{author}}(a) = \texttt{\&fname}(a) \bullet \texttt{\&lname}(a)$$

where `author` is the type name from $\mathcal{S}_1$, a is a variable of sort `author` from $\mathcal{S}_1$, `&fname` and `&lname` are selector functions for the `author` product type in $\mathcal{S}_1$, and $\bullet$ is the append operation on lists of strings (using the second list polytype in Example 1 instantiated with `String`, which therefore becomes a subsort of `List`).

The converse translation for these schemas involves both conditions and semantic functions,

$$\begin{array}{llll}
\mathcal{M}'_{\texttt{author}}(a) = a & \texttt{if}\ |a| = 1 & \qquad \mathcal{M}'_{\texttt{author}}(a) = \texttt{tail}(a) & \texttt{if}\ |a| = 2\\
\mathcal{M}'_{\texttt{author}}(a) = \perp & \texttt{if}\ |a| = 1 & \qquad \mathcal{M}'_{\texttt{author}}(a) = \texttt{head}(a) & \texttt{if}\ |a| = 2
\end{array}$$

where a is a variable of type `author` from $\mathcal{S}_2$ and $\perp$ is a null value. (Formally, each pair of equations should be one equation with a polymorphic conditional operation.) □

The first step to schema morphisms is defining a joint explicit type algebra $T^{\#\#}_{\mathcal{S},\mathcal{S}'}$ for schemas $\mathcal{S}, \mathcal{S}'$ of the same species: it is the free unsorted algebra with constants the elements of N, N' and S^D; with unary function symbols $\texttt{\#}n$ for $n \in N$ and $\texttt{\#}n'$ for $n' \in N'$; plus all the (unsorted) type constructors for that species. Now let $T^{\#}_{\mathcal{S},\mathcal{S}'}$ be the quotient of $T^{\#\#}_{\mathcal{S},\mathcal{S}'}$ by the type equations of both $\mathcal{S}$ and $\mathcal{S}'$; it is the **joint explicit reduced type algebra**, and its elements will serve as sorts for an algebra of possible translation functions.

The basic translation algebra $T_{\mathcal{S},\mathcal{S}'}$ is defined as follows: (1) its sorts are the complete elements of $T^{\#}_{\mathcal{S},\mathcal{S}'}$; (2) its operations are those of $\mathcal{C}_{\mathcal{P}}$ instantiated with the sorts of $T_{\mathcal{S},\mathcal{S}'}$; and (3) its equations are those of E^D, plus $\mathcal{C}_{\mathcal{P}}$ instantiated with the sorts of $T_{\mathcal{S},\mathcal{S}'}$. Given a set P and a function $\mu\colon\ P \to T^{\#}_{\mathcal{S},\mathcal{S}'}$, let $\texttt{\%}P$ be a set of new constants $\texttt{\%}p$ of sort $\mu(p)$ for $p \in P$, and given a function $\mathcal{M}$ defined on P, let $\texttt{\%}\mathcal{M}$ be a set of new unary function symbols $\texttt{\%}\mathcal{M}_p$ of argument sort $\mu(p)$ for $p \in P$. Now we are ready for the main concept of this paper, which uses the extnesion $T_{\mathcal{S},\mathcal{S}'}(\texttt{\%}P, \texttt{\%}\mathcal{M})$ of $T_{\mathcal{S},\mathcal{S}'}$ by the symbols in $\texttt{\%}P$ and $\texttt{\%}\mathcal{M}$:

Definition 11. Given a schema species, a **schema morphism** of that species from $\mathcal{S}\colon N \to T(N)$ to $\mathcal{S}'\colon N' \to T(N')$ consists of a set P containing N, a function $\mu\colon P \to T^{\#}_{\mathcal{S},\mathcal{S}'}$ such that $\mu(n) = n$ for $n \in N$, and a function $\mathcal{M}\colon P \to T_{\mathcal{S},\mathcal{S}'}(\%P, \%\mathcal{M})$ such that each $\mathcal{M}(p)$ has just one free variable $\%p$, of sort $\mu(p)$, and such that $\mathcal{M}(Top)$ has sort Top'; we may also use the notations $\mathcal{M}_p$ and $\mathcal{M}_p(\%p)$. A schema morphism $\mathcal{M}\colon \mathcal{S} \to \mathcal{S}'$ is **simple** if the terms $\mathcal{M}_p$ contain no operations from Σ^D, no selectors from $\mathcal{S}'$, and no constructors from $\mathcal{S}$.

A schema morphism $\mathcal{M}$ defines a **match relation** $match_{\mathcal{M}}$ on $N \times N'$ by $match_{\mathcal{M}}(n, n')$ iff the term $\mathcal{S}'(n')$ occurs (modulo equations) in $\mathcal{M}_p$, in which case we say that n and n' **match**. $\Box$

Intuitively, a schema morphism $\mathcal{M}\colon \mathcal{S} \to \mathcal{S}'$ describes a way to select structures from $\mathcal{S}$ databases and then put them together to form $\mathcal{S}'$ databases. The function symbols $\%\mathcal{M}_p$ allow recursion in schema morphism definitions, since they will be replaced by the terms $\mathcal{M}_p$; they also allow the use of auxiliary functions. Schema morphisms may have semantic functions, conditions and null values, because terms in $T_{\mathcal{S},\mathcal{S}'}$ can contain operations on data (from D) as well as conditionals (e.g., `if_then_else_`) and nulls; however, these are excluded from simple schema morphisms.

Definition 12. The **rewrite rules** of a schema morphism $\mathcal{M}$ from $\mathcal{S}$ to $\mathcal{S}'$ are: $\%p \mapsto \mathcal{M}_p(\%p)$ for all $p \in P$, where $\%p$ matches terms of type $\mu(p)$ (and in particular, $\%n$ matches terms of type n for $n \in N$); and $\%\mathcal{M}_p(\%p) \mapsto \mathcal{M}_p(\%p)$ for all $p \in P$. The **translation** of an $\mathcal{S}$ database B to a $\mathcal{S}'$ database, denoted $\overline{\mathcal{M}}(B)$, is obtained by successively applying[4] the rewrite rules of $\mathcal{M}$ to B, and evaluating any S^D-sorted terms in D. The same procedure may be applied to substructures B_0 of $\mathcal{S}$ databases, for the result of which we write $\overline{\mathcal{M}}_n(B_0)$, where B_0 has type n; we may also drop the overbar from $\overline{\mathcal{M}}$. Under these conventions, $\overline{\mathcal{M}} = \mathcal{M}_{Top}$.

A schema morphism $\mathcal{M}\colon \mathcal{S} \to \mathcal{S}'$ is **correct** iff $\overline{\mathcal{M}}(B)$ conforms to $\mathcal{S}'$ whenever B conforms to $\mathcal{S}$. If $(\mathcal{S}, \mathcal{C})$ and $(\mathcal{S}', \mathcal{C}')$ are constrained schemas of the same species, then a correct morphism $\mathcal{M}\colon \mathcal{S} \to \mathcal{S}'$ is a **constrained schema morphism** if $B \models (\mathcal{S}, \mathcal{C})$ implies $\overline{\mathcal{M}}(B) \models (\mathcal{S}', \mathcal{C}')$. $\Box$

From now on, we shall consider correctness to be part of the definition of schema morphism.

Example 17. The schema morphism $\mathcal{M}\colon \mathcal{S}_1 \to \mathcal{S}_2$ of Example 16 is obtained by using tupling for the product constructor, and using iteration for the semantic function over list elements to the list constructor. This yields the following formulae, where B is a database conforming to $\mathcal{S}_1$

$$\begin{aligned}
\mathcal{M}_{\texttt{Bib}}(B) &= \bullet_{K\in B}\mathcal{M}_{\texttt{Book}}(K)\\
\mathcal{M}_{\texttt{Book}}(K) &= \langle \mathcal{M}_{\texttt{Title}}(p_1(K)), \mathcal{M}_{\texttt{Year}}(p_2(K)), \mathcal{M}_{\texttt{Author}}(p_3(K))\rangle\\
\mathcal{M}_{\texttt{Author}}(A) &= \bullet_{a\in A}\mathcal{M}_{\texttt{author}}(a)\\
\mathcal{M}_{\texttt{author}}(a) &= \texttt{\&fname}(a) \bullet \texttt{\&lname}(a)
\end{aligned}$$

[4] Because of the form of morphisms, this process is necessarily terminating and Church-Rosser, and hence (by a basic theorem of term rewriting theory) always produces a unique result.

where $\bullet$ is the iterated list insert operation, and where %s are omitted from %$\mathcal{M}$ operations on the right sides. Putting all the formulae together yields the following master formula,

$$\mathcal{M}_{\texttt{Bib}}(B) = \bullet_{K \in B} \langle p_1(K), p_2(K), \bullet_{a \in p_3(K)} p_1(a) \bullet p_2(a) \rangle \ .$$

Note that the variables K, A, a do not strictly speaking occur in this formula, but are just part of a conventional notation for iterators. The last formula can be used directly to translate databases in the $\mathcal{S}_1$ format to databases in the $\mathcal{S}_2$ format. It is unfortunate that the formula looks so complex, because for the higher level types `Book` and `Author`, all it does is copy over and rebuild structures; the only changes are for the type `author`. But as with queries, although this notation is not so good for users, it is good for theory, and our SCIA tool, briefly described in Section 3.7, provides a very nice graphical interface for helping users define schema morphisms that avoids complex notation. □

Example 18. The morphism $\mathcal{M}$ below removes year data from bibliographic databases that conform to the schema of Example 6; it raises an interesting issue, which we then show how to resolve.

$$\begin{aligned}
\mathcal{M}_{\texttt{Bib}}(B) &= \bullet_{K \in B} \mathcal{M}_{\texttt{Book}}(K) \\
\mathcal{M}_{\texttt{Book}}(K) &= \langle \mathcal{M}_{\texttt{Title}}(p_1(K)), \mathcal{M}_{\texttt{Author}}(p_3(K)) \rangle \\
\mathcal{M}_{\texttt{Title}}(T) &= T \\
\mathcal{M}_{\texttt{Author}}(A) &= A \\
\mathcal{M}_{\texttt{Year}}(Y) &= Y
\end{aligned}$$

One might think the last equation is not needed, because year data is not being translated. But the definition of schema morphism requires that $\mathcal{M}$ be a total function on N, so something must be done. However, this raises another problem, because the definition of morphism requires $\mathcal{M}_{\texttt{Year}}(Y)$ to be a term of a sort in the target schema. A simple solution to this dilemma is to add `Year` to the target as an unreachable name, with $\mathcal{S}'(Y) = \texttt{Int}$; the resulting schema is then r-equivalent (see Definition 13 just below) to the original, and $\mathcal{M}$ is a constrained schema morphism for it. □

Definition 13. Two schemas of the same species are **r-equivalent** iff they are equal as functions when restricted to their reachable names. □

Proposition 1. Every schema $\mathcal{S} \colon N \to T(N)$ is r-equivalent to a unique reachable schema $\mathcal{S} \colon N_* \to T(N_*)$, and r-equivalent schemas have exactly the same databases. □

We say that schema morphisms (of the same species, between the same schemas) are **translation equivalent** iff they define the same transformation on databases. The following makes available many powerful concepts and results from category theory:

Theorem 1. For a given species, its schemas and its schema morphisms modulo translation equivalence form a category, where the identity morphism $1_{\mathcal{S}}$ on $\mathcal{S}$ is given by $(1_{\mathcal{S}})_n(\%n) = \%n$, and where the composition of morphism $\mathcal{M} \colon P \to T_{\mathcal{S},\mathcal{S}'}$ with $\mathcal{M}' \colon P' \to T_{\mathcal{S}',\mathcal{S}''}$ has index set $Q = P \uplus P'$ (disjoint union), and is defined by

$(\mathcal{M};\mathcal{M}')_p \ = \mathcal{M}_p$ for $p \in P$
$(\mathcal{M};\mathcal{M}')_{p'} = \mathcal{M}'_{p'}$ for $p' \in P'$
Let $\mathbb{S}ch(D,\mathcal{P},\mathcal{R})$ denote the category of all schemas of the species $(D,\mathcal{P},\mathcal{R})$. □
The proof is straightforward, but note that the identity laws would fail without translation equivalence. Due to the generality allowed for $\mathcal{R}$, the categories $\mathbb{S}ch(D,\mathcal{P},\mathcal{R})$ do not necessarily have colimits, although the familiar practical cases admit finite colimits.

Theorem 1 immediately gives the following: schemas $\mathcal{S}_1$ and $\mathcal{S}_2$ are **isomorphic** if there are morphisms $\mathcal{M}\colon \mathcal{S}_1 \to \mathcal{S}_2$ and $\mathcal{M}'\colon \mathcal{S}_2 \to \mathcal{S}_1$ such that $\mathcal{M};\mathcal{M}'$ and $\mathcal{M}';\mathcal{M}$ are both (translation equivalent to) identity morphisms. For example, $\mathcal{S}_1, \mathcal{S}_2$ of Example 16 are not isomorphic, because $\mathcal{S}_1$ has more structure. As an exercise, the reader might find the compositions $\mathcal{M};\mathcal{M}'$ and $\mathcal{M}';\mathcal{M}$ of the morphisms in Example 16; one is an identity, the other is not. Schema isomorphism is structural equivalence (in the sense of programming language theory).

It cannot be expected that every kind of constraint can be translated by schema morphisms, even for relatively familiar and simple species. For example, the morphism in Example 18 that drops year data from the bibliographic schema of Example 6 cannot preserve the constraint that there cannot be two books with the same title and authors in the same year (this constraint might be used to force new editions of a book to be at least one year later).

Theorem 1 also provides the correct notions of product, sum, and more generally, limit and colimit, for schemas. All these concepts apply to any species of schema, and colimits give an elegant and extremely general notion of heterogeneous schema integration.

Definition 14. Given a species of schema, a family of constraints for schemas of that species is **translatable** iff for any morphism $\mathcal{M}\colon \mathcal{S} \to \mathcal{S}'$ and any constraint set $\mathcal{C}$ of that family for $\mathcal{S}$, there exists a constraint set $\mathcal{C}'$ of that family for $\mathcal{S}'$ such that for any database B of $\mathcal{S}$, we have $B \models_{\mathcal{S}} \mathcal{C}$ iff $B' \models_{\mathcal{S}'} \mathcal{C}'$ where B' is r-equivalent to $\mathcal{M}(B)$. An **effective** translatable constraint family includes an effective procedure for computing $\mathcal{C}'$ from $\mathcal{M}$ and $\mathcal{C}$, in which case we may write $\mathcal{M}(\mathcal{C})$ for $\mathcal{C}'$.

A translatable constraint family is **composable** iff whenever $\mathcal{C}$ is a constraint set of that family for $\mathcal{M}$ and $\mathcal{M}'\colon \mathcal{S}' \to \mathcal{S}''$, then $\mathcal{M}'(\mathcal{M}(\mathcal{C}))$ and $(\mathcal{M};\mathcal{M}')(\mathcal{C})$ are semantically equivalent, in the sense that a $\mathcal{S}''$ database B'' satisfies one iff it satisfies the other. A constraint family is a **suitable** family for a species iff it is composable and effective translatable. A **constrained species** consists of a schema species $(D,\mathcal{P},\mathcal{R})$ plus a suitable family $\mathcal{F}$ of constraints for that species, written $(D,\mathcal{P},\mathcal{R},\mathcal{F})$; a constrained schema of constrained species $(D,\mathcal{P},\mathcal{R},\mathcal{F})$ is a constrained schema over $(D,\mathcal{P},\mathcal{R})$ that uses only constraints from $\mathcal{F}$. □
Theorem 2. The constrained schemas of a given constrained species form a category under the composition of Theorem 1. □
For constrained schemas, isomorphism implies that the constraint sets are semantically equivalent. An interesting research programme is to determine what constraint families are suitable for various important schema species. The following is a beginning:

Theorem 3. Data integrity constraints are a suitable family for the essential XML schema species with simple schema morphisms. □

Constraints for the relational species have been studied in many papers, including [32, 38, 30], and it seems that results in [30] can be lifted from the level of database translations to that of schema morphisms, and given an elegant formulation with final morphisms in a 2-category of schemas, though details will have to wait for a future paper.

3.7 A Schema Morphism Tool

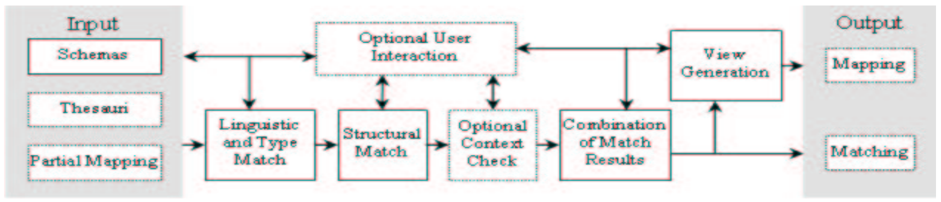

Fig. 1. Overview of the Mapping Process

Our laboratory at UCSD is designing and buiding a GUI-based interactive tool called SCIA, for translating, integrating and querying data with associated DTD or XML Schema metadata [36, 42]. Since experience shows that fully automatic schema morphism generation is infeasible, SCIA attempts to minimize total user effort by identifying critical decision points, where a small user input can yield a large reduction of future effort. More precisely, a **critical point** is where a core context has either no good matches, or has more than one good match, where **core contexts** are the most important contextualizing elements for tags within their subtrees [42]; core contexts are determined by heuristics and/or user input, and typically have a large subtree. In interactive mode, the tool solicits user input at critical points, and iterates until both the user and the tool are satisfied; in automatic mode, it does just one pass using default strategies.

Each pass has five steps: linguistic and data type matching; structural matching; context checking; combining match results; and view generation. Figure 1 depicts the mapping process for one source and one target schema. Other tools only do schema matching, and in fact, only find the easiest matches, leaving the most difficult matches for the user to do by hand [39]; semantic functions and conditions are not treated, or else are left for the user to supply using a different view generation tool, whereas our tool integrates all these features. A major experimental result is that using critical points can significantly reduce total user effort [36]. As far as we know, SCIA is the only tool to support complex matches having semantic functions and conditions, as well as the only tool to utilize critical points. Planned extensions to SCIA are discussed in Section 5.

4 Institutions, Ontologies and Heterogeneous Integration

Institutions arose in computer science in response to the explosion of logics being used there, and the desire to do as much theoretical computer science as possible, independently of the choice of logic [20]. In particular, a general collection of operations for structuring theories was developed for applications to modularization in languages for specification, programming, and knowledge representation, which also applies to ontologies [14, 15, 28, 25, 18].

Institutions axiomatize the notion of logical system by extending Tarski's idea that the *satisfaction* of a sentence by a model is fundamental. The extension parameterizes models, sentences, and satisfaction over signatures, rather than assuming a single fixed signature, as Tarski did. Institutions can be seen as a theory of truth and proof that is relativized by context, where contexts are represented by objects in the category of signatures; these need not be anything like the signatures used in logic, since they can come from any category at all. Institutions have been successfully applied to give semantics for powerful module systems [25], and to multi-logic specification languages [10], as well as to databases [1, 18], and to behavioral types and semantics for the object paradigm [21, 16]; a recent programme of Diaconescu seeks to generalize as much classical model theory as possible, e.g., Craig interpolation [11] and Beth definability.

This section uses institutions for two main purposes: to handle the heterogeneity of logics used for ontologies, and to handle translation and integration of databases over heterogeneous schema species. Section 4.3 discusses ways attach ontologies to databases. The machinery is a bit heavy, but the resulting generality is great, and the applications important.

4.1 Institutions and their Morphisms

An ontology is just a theory over a logic, i.e., a set of sentences in that logic. Integrating ontologies raises issues analoguous to those discussed in Section 2.1: Morphisms of ontologies over a single logic are well enough understood, so that co-cones and colimits of ontologies can be used (see [1–3, 18]), but to integrate ontologies over different logics, the notion of logic must be formalized, along with morphisms of theories over different logics, for which morphisms of logics are also needed. Such issues can be addressed using institutions. Here is the formal definition, in which $\mathbb{S}et$ denotes the category of sets and functions:

Definition 15. An **institution** consists of an abstract category $\mathbb{S}ign$ of signatures, a functor $Sen\colon \mathbb{S}ign \to \mathbb{S}et$ for sentences, a functor $Mod\colon \mathbb{S}ign^{op} \to \mathbb{S}et$ for models, and a satisfaction relation $\models_\Sigma$ between models and sentences such that for every signature morphism $f\colon\ \Sigma \to \Sigma'$, we have $f(M) \models_{\Sigma'} e$ iff $M \models_\Sigma f(e)$, for every Σ-model M and every Σ'-sentence e, where $f(M)$ abbreviates $Mod(f)(M)$ and $f(e)$ abbreviates $Sen(f)(e)$. □

This version of the institution notion does not include proofs. However, it is natural to consider proofs as morphisms of sentences, so that the target of Sen

becomes the category $\mathbb{C}at$ of small categories. Similarly, homomorphisms of models can be included by letting the target of Mod be $\mathbb{C}at$. More detail can be found in [20] and especially [35].

Example 19. Equational logic as described in Section 3.1 is a relatively simple institution: its signatures are as defined there; its Σ-sentences are Σ-equations; its models are Σ-algebras; and satisfaction is as described there. First order logic is quite similar: for the simplest unsorted case with only predicates, signatures are sets of predicate symbols with their arities, Σ-sentences are the usual first order sentences over Σ, Σ-models are the usual first order structures, and satisfaction is the usual Tarskian semantic truth. See [20] for details of these and several other institutions for formal logics. □

A careful argument that every logic can be represented by an institution is given in [35]; because in general more than one institution represents a given logic, a notion of logical equivalence of institutions is also an important part of that development. Institutions arise from databases in several different ways, some of which are described in [19]. The following is the most straightforward in the context of this paper:

Theorem 4. Given a schema species, an institution is formed with the schemas $\mathcal{S}$ of that species as signatures, with a class of suitable constraints as $\mathcal{S}$-sentences, with $\mathcal{S}$-databases as $\mathcal{S}$-models, and with satisfaction as in Definition 10. □

In database theory, schema mappings and the database translations that they induce go in the same direction. Because this is opposite to what institutions do, we use the ***opposite*** of the category of sets for the target of the model functor in the institution of Theorem 4 (technically, this gives a "variant institution" in the sense of [24])[5]. For databases, institutional duality between sentences and databases was first discussed in [1], but Theorem 4 is more precise and more general, and differs from [1] in that its constraints and databases are both covariant.

The following allows us to treat ontologies over arbitrary logics, and also provides some very useful further concepts:

Definition 16. A **theory** over an institution $\mathcal{I}$ is a pair (Σ, E) where E is a set of Σ-sentences. A Σ-model M **satisfies** (Σ, E) iff $M \models_\Sigma e$ for all $e \in E$. The **model class** of a theory $(\Sigma, E)^*$ is the class of all models that satisfy that theory, and the **theory** $\mathcal{M}^*$ of a class $\mathcal{M}$ of models is the class of all sentences that are satisfied by all models in $\mathcal{M}$. This situation is a *Galois connection*, which gives us notions of **closed theory**, i.e., such that $(\Sigma, E)^{**} = (\Sigma, E)$, and **closed model class**.

A **theory morphism** $(\Sigma, E) \to (\Sigma', E')$ over $\mathcal{I}$ is a signature morphism $f\colon \Sigma \to \Sigma'$ such that $f(E) \subseteq E'^{**}$. □

The following is an important result from [20]:

Theorem 5. Theories and their morphisms over a fixed institution $\mathcal{I}$ form a category denoted $\mathbb{T}h(\mathcal{I})$, which has whatever colimits $\mathbb{S}ign$ has.

[5] Essentially all properties of institutions carry over to variant institutions, but the details are a bit technical and are omitted here.

A significant benefit of this result is a powerful method for structuring ontologies into modules, including inheritance and sums of modules, shared submodules, and (most usefully) modules parameterized by other modules, in the style made popular by the ML programming language, but originating in the Clear specification language [9], and further developed under the name **parameterized programming** [14]; such features are important for structuring large and/or complex theories (or programs) to better support reuse and reasoning.

In order to get the same powerful module system for ontologies over heterogeneous logics, we need to be able to translate between different logics. The following provides a basis for this:

Definition 17. An **institution morphism** from $\mathcal{I}$ to $\mathcal{I}'$ consists of a functor $\Phi\colon \mathbb{S}ign \to \mathbb{S}ign'$ and two natural transformations, $\alpha_\Sigma\colon Sen(\Sigma) \to Sen'(\Phi(\Sigma))$ and $\beta_\Sigma\colon Mod'(\Phi(\Sigma)) \to Mod(\Sigma)$, such that $M' \models_{\Phi(\Sigma)} \alpha_\Sigma(e)$ iff $\beta_\Sigma(M') \models_\Sigma e$, for all signatures Σ of $\mathcal{I}$, all $\Phi(\Sigma)$-models M' of $\mathcal{I}'$, and all Σ-sentences e of $\mathcal{I}$. An institution $\mathcal{I}$ is a **subinstitution** of another $\mathcal{I}'$ if Φ and each α_Σ are inclusions, and the β_Σ are forgetful functors. Institutions with institution morphisms form a category, denoted $\mathbb{I}ns$. $\square$

Several other notions of institution morphism, and properties of the resulting categories, are discussed in [24].

4.2 Heterogeneous Integration

Situations in which one kind of structure is indexed by another are common in mathematics, computer science, and their applications, and are the essence of many information integration problems. Alexander Grothendieck developed a very general way to deal with this kind of structural heterogeneity, as part of his brilliant reformulation of algebraic geometry into the language of category theory, in order to solve a number of important outstanding problems. "Structures" are of course formalized by categories with their structure-preserving morphisms, and indexing is given by a functor $F\colon I^{op} \to \mathbb{C}at$. **Grothendieck flattening** turns an indexed family of categories into a single category. This is an elegant and very general way to deal with the many kinds of heterogeneity that can be considered to arise from an **indexed family**, which is a functor $F\colon \mathbb{I}^{op} \to \mathbb{C}at$, assigning to each object i in the index category $\mathbb{I}$ a structure represented by the category $F(i)$.

Definition 18. The **Grothendieck category** of an indexed family $F\colon \mathbb{I}^{op} \to \mathbb{C}at$, denoted $\mathbb{G}r(F)$, has as its objects pairs (i, A) where i is an object in $\mathbb{I}$ and A is an object in $F(i)$, and has as its morphisms $(i, A) \to (i', A')$ pairs (f, h) where $f\colon i \to i'$ in $\mathbb{I}$ and $h\colon A \to F(f)(A')$ in $F(i)$; we call such morphisms **heteromorphisms**. Given also $(f', h')\colon (i', A') \to (i'', A'')$, define the **composition** $(f, h); (f', h')\colon (i, A) \to (i'', A'')$ to be $(f; f',\ h; F(f)(h'))$. $\square$

It is easy to check that this gives a category. Several useful results about colimits and limits in Grothendieck categories[6] are given in [41], although a better exposition is given in Section 2.1 of [24].

[6] Gray [31] gives the basic results for a much more general notion of indexed category, but this extra complexity is not needed for this paper.

Example 20. As a first application of Grothendieck flattening, we build the category of all algebras over all signatures; for simplicity, we treat only the unsorted case. The index category $\mathbb{I}$ is the category $\mathbb{AS}ign$ of algebraic signatures, which consist of a set Σ of operation symbols with an arity function $\alpha\colon \Sigma \to \omega$ (where ω is the set of natural numbers), and where a morphism from Σ to Σ' is a function $f\colon \Sigma \to \Sigma'$ such that $\alpha'(f(\sigma)) = \alpha(\sigma)$ for all $\sigma \in \Sigma$. Now let $F(\Sigma)$ be the category $\mathbb{A}lg(\Sigma)$ of all Σ-algebras with Σ-homomorphisms. The reader may check the functoriality of $F\colon \mathbb{AS}ign^{op} \to \mathbb{C}at$.

Then an **algebra heteromorphism**, or cryptomorphism, $(\Sigma, A) \to (\Sigma', A')$ is (f, h) where $f\colon \Sigma \to \Sigma'$ and $h\colon A \to F(f)(A')$ in $\mathbb{A}lg(\Sigma)$. It is not difficult, using general results (relatively easy expositions are given in [41, 24]) to show that $\mathbb{G}r(\mathbb{A}lg)$ has both limits and colimits; this allows integrating algebras over different signatures. □

Similarly, we can deal with the logical heterogeneity of multiple institutions with the following Grothendieck theory construction, which simplifies [10]:

Definition 19. Given a category $\mathbb{O}$ of institutions, let $\mathbb{GT}h(\mathbb{O})$ have objects $(\mathcal{I}, \Sigma, E)$ where $\mathcal{I}$ is in $\mathbb{O}$ and (Σ, E) is a theory of $\mathcal{I}$, and have **morphisms** from $(\mathcal{I}, \Sigma, E)$ to $(\mathcal{I}', \Sigma', E')$ consisting of an institution morphism $(\Phi, \alpha, \beta)\colon \mathcal{I} \to \mathcal{I}'$ in $\mathbb{O}$ and a signature morphism $f\colon \Sigma' \to \Phi(\Sigma)$ in $\mathcal{I}'$ such that $E \subseteq \alpha_\Sigma(f(E'))^{**}$. Composition is defined as for heteromorphisms in the general Grothendieck flattening construction, making $\mathbb{GT}h(\mathbb{O})$ into a category. □

In fact, $\mathbb{GT}h(\mathbb{O})$ is the Grothendieck category of the theory functor $Th\colon \mathbb{O} \to \mathbb{C}at$; it is also the theory category of the Grothendieck institution [10] of $\mathbb{O}$ viewed as a diagram, i.e., a (contravariant) functor from a small category to the category of institutions. The Grothendieck institution of [10] is a remarkable construction of a single institution from an indexed family of institutions; its category of signatures is the Grothendieck category of the indexed category of signatures of the institutions involved, and similarly for its models and sentences. The Grothendieck institution construction also tends to lift significant logical properties from individual institutions to the whole, e.g., Craig interpolation [11]. If $\mathbb{O}$ satisfies some reasonable conditions, then $\mathbb{GT}h(\mathbb{O})$ is cocomplete (because it is the theory category of an institution, or more directly, using methods in [24, 41]).

We are now in a position to apply Grothendieck constructions to our abstract database theory. The first step is to extend Theorem 2 to all the schemas and databases of a given species, using some suitable class of constraints:

Theorem 6. Applying the Grothendieck theory construction to the categories of Theorem 2 using schema morphisms over a given schema species yields a category of all suitably constrained schemas over that species. □

We can also apply the construction of Definition 19 to the institutions of Theorem 4, to obtain an heterogeneous institution of constrained schemas over constrained species. To further extend integration to all possible species, we need the following:

Definition 20. Given schema species $(D, \mathcal{P}, \mathcal{R})$ and $(D', \mathcal{P}', \mathcal{R}')$, a **species morphism** $(D, \mathcal{P}, \mathcal{R}) \to (D', \mathcal{P}', \mathcal{R}')$ consists of an injective algebra heteromorphism $D \to D'$ and a theory morphism $\mathcal{P} \to \mathcal{P}'$ such that for any function $f\colon N \to N'$, if $t \in T(N)$ and[7] $\mathcal{R}(n, t)$ then $\mathcal{R}'(f(n), t')$, where t' is the image in $T'(N')$ of t under the map $T(N) \to T(N')$ induced by the maps $D \to D'$ and $\mathcal{P} \to \mathcal{P}'$ (this exists because $T(N)$ is the initial algebra generated by D, $\mathcal{P}$ and N). $\Box$

It is not difficult to check the following:

Proposition 2. Species with species morphisms form a category. $\Box$

Definition 21. Given constrained species $(D, \mathcal{P}, \mathcal{R}, \mathcal{F})$ and $(D', \mathcal{P}', \mathcal{R}', \mathcal{F})$ a **morphism** consists of a species morphism $(D, \mathcal{P}, \mathcal{R}) \to (D', \mathcal{P}', \mathcal{R}')$ plus an effective procedure for translating $\mathcal{F}$ constraints to $\mathcal{F}'$ constraints. $\Box$

Theorem 7. Applying Grothendieck flattening to the categories of Theorem 6 using schema species morphisms, yields a category of suitably constrained schemas over all species. $\Box$

Now we can use colimits of co-cones (i.e., co-relations) for the global-as-view case, or limits of cones for the local-as-view case (see Section 2.1 for these concepts), to integrate schemas of heterogeneous species, which are pairs $(\mathcal{E}, \mathcal{S})$, where $\mathcal{S}$ is a schema of species $\mathcal{E}$. The index category is the category of all species with their morphisms. By using the Grothendieck institution construction instead of the Grothendieck theory and flattening constructions, we could get an institution with heterogeneous databases as well as heterogeneous schemas, but we have preferred to avoid the technicalities that would be required for this.

An alternative approach to ontology semantics, e.g., in [33], uses the information[8] flow theory of Barwise and Seligman [2], with its classifications, infomorphisms, and local logics. However, classifications are actually a special case of institutions, where the category of signatures has just one object and one morphism, sets of tokens are models, types are sentences, classification is satisfaction, and the infomorphisms are institution (co)morphisms. Moreover, local logics form a pleasant institution with classifications as models. See [19, 18] for details, including a number of new results about information flow theory that arise from using a categorical framework.

4.3 Ontologies for Databases

The motivation for attaching an ontology to a dataset is to enable more powerful query and integration capabilities for that data; this can be done in several different ways [19, 8]. Perhaps the simplest are based on the observation that ontologies can be seen as schemas; this also can be done in several ways, as illustrated in the following:

[7] Recall that $\mathcal{R}(n, t)$ means that t is admissible for n under $\mathcal{R}$.

[8] The Barwise and Seligman [2] use of the word "information" is consistent with their realist position, that information exists independently of human beings, whereas this paper takes a constructivist position and therefore uses the word "data."

Example 21. A schema $\mathcal{S}\colon N \to T(N)$ of a given species can be extended with an ontology, by adding to N new constants `OntoData`, `Onto`, and $\texttt{BRel}_i$ for $i = 1, ..., M$, letting `OntoData` be the new top, renaming the old top of $\mathcal{S}$ to `Data`, and adding the following to the definition of $\mathcal{S}$,

$$\begin{aligned} &\mathcal{S}(\texttt{OntoData}) = \texttt{Onto} \times \texttt{Data} \\ &\mathcal{S}(\texttt{Onto}) \ \ = \texttt{BRel}_1 \times ... \times \texttt{BRel}_M \\ &\mathcal{S}(\texttt{BRel}_i) \ \ = \texttt{SetOf}(\texttt{F}_{i1} \times \texttt{F}_{i2}) \end{aligned}$$

where $\texttt{F}_{ij}$ are in S^D. Note that in general this may require modifying the species of $\mathcal{S}$. Database instances of the extended schema will instantiate the $\texttt{F}_{ij}$ with unique identifiers. This is essentially an encoding of RDF notation, and typical `BRel`'s would be `is_a` and `has_a`. It is also possible to impose axioms on relations with constraints, such as transitivity of `is_a`,

$$(\forall X, Y, X) \ \ \texttt{is\_a}(X, Y) \text{ and } \texttt{is\_a}(Y, Z) \text{ implies } \texttt{is\_a}(X, Z)\,,$$

where the variables X, Y, Z are of the universal (maximum) sort. A perhaps subtle point is that axioms in ontologies are meant to be used to support the use of reasoning, rather than to constrain what must be in the schema; that is, axioms like the above generate "virtual" database entries, instead of requiring that they are actually present.

RDF triples can also be encoded more directly, with

$$\begin{aligned} &\mathcal{S}(\texttt{Onto}) = \texttt{SetOf}(\texttt{Triple}) \\ &\mathcal{S}(\texttt{Triple}) = \texttt{String} \times \texttt{URI} \times \texttt{URI} \end{aligned}$$

but this is less explicit about the relations involved, and also requires some way to connect URI's the unique identifiers in databases.

These declarations could be placed into the species from the beginning, which would require that every database of that species has an ontology; in this case, schema morphisms would also have to translate ontologies. We could even encode one particular ontology into the species definition. Yet another approach is to have a separate species of ontologies, which are then related to database schemas by heteromorphisms. □

It is interesting to ask to what extent more expressive ontology languages can be encoded as schemas, so that tools like SCIA can be used for ontology integration. The full generality of institutions is certainly too great, but perhaps one can abstractly characterize the institutions such that their theories can be encoded as constrained schemas.

It may be interesting to notice that `is_a` and `has_a` are inherent to order sorted algebra, and in fact have been used in our formalization of database schemas: the subsort hierarchy is an `is_a` hierarchy, and the subterm relation defines a `has_a` hierarchy that is correctly related to the `is_a` hierarchy [23].

5 Conclusions and Future Work

We are extending our SCIA tool (Section 3.7) to handle additional schema species, such as spreadsheet, object oriented, etc. This is relatively straightforward for schema matching and mapping, because it is only necessary to write a preprocessor to convert schemas into the internal form of the tool (which consists of a tree and a graph in RDF notation). We have done this for the relational species, but additional effort is needed for data transformation and integration, especially writing new view generation code. A next step is to extend the tool to make use of ontologies, using the semantics developed in this paper. We will also explore applications to workflows.

Theoretical issues that need further investigation include the properties of various categories discussed above, including schemas and constrained schemas over a fixed species, and the various Grothendieck categories. It would be good to simplify the definition of schema morphism, if possible. Relationships to approaches based on local logics [2] should also be explored further than in [19]. It would also be interesting to determine suitable constraint families for various species of schema, under various subcategories of schema morphisms.

References

1. Suad Alagic and Philip Bernstein. A model theory for generic schema management. In Giorgio Ghelli and Gosta Grahne, editors, *Proc. Database Programming Languages 2001*, pages 228–246. Springer, 2002. Lecture Notes in Computer Science, volume 2397.
2. Jon Barwise and Jerry Seligman. *Information Flow: Logic of Distributed Systems.* Cambridge, 1997. Tracts in Theoretical Computer Science, vol. 44.
3. Trevor Bench-Capon and Grant Malcolm. Formalising ontologies and their relations. In *Proceedings of the 16th International Conference on Database and Expert Systems Applications (DEXA '99)*, pages 250–259. Springer, 1999. Lecture Notes in Computer Science, volume 1677.
4. Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.
5. Philip Bernstein. Applying model management to classical metadata problems. In *Proc. Conf. on Innovative Database Research*, pages 209–220, 2003.
6. Philip Bernstein. Model management webpage, 2003. At `http://research.microsoft.com/db/ModelMgt/`.
7. Philip Bernstein and Erhard Rahm. Data warehouse scenarios for model management. In *Entity-Relation Conference (ER 2000)*, volume 1920 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2000.
8. Shawn Bowers and Bertram Ludäscher. An ontology-driven framework for data transformation in scientific workflows. In *Proc. Data Integration in the Life Sciences 2004*, pages 1–16. Springer, 2004.
9. Rod Burstall and Joseph Goguen. Putting theories together to make specifications. In Raj Reddy, editor, *Proceedings, Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058. Department of Computer Science, Carnegie-Mellon University, 1977.
10. Răzvan Diaconescu. Grothendieck institutions. *Applied Categorical Structures*, 10:383–402, 2002.
11. Răzvan Diaconescu. Interpolation in Grothendieck institutions. *Theoretical Computer Science*, 311:439–461, 2004.

12. Paul Dourish. What we talk about when we talk about context. *Personal and Ubiquitous Computing*, 8(1):19–30, December 2003. Online edition.
13. Mary Fernandez, Jérôme Siméon, and Philip Wadler. A semi-monad for semi-structured data. In *Proc. ICDT'01*, pages 263–300, 2001.
14. Joseph Goguen. Principles of parameterized programming. In Ted Biggerstaff and Alan Perlis, editors, *Software Reusability, Volume I: Concepts and Models*, pages 159–225. Addison Wesley, 1989.
15. Joseph Goguen. Higher-order functions considered unnecessary for higher-order programming. In David Turner, editor, *Research Topics in Functional Programming*, pages 309–352. Addison Wesley, 1990. University of Texas at Austin Year of Programming Series; preliminary version in SRI Technical Report SRI-CSL-88-1, January 1988.
16. Joseph Goguen. Types as theories. In George Michael Reed, Andrew William Roscoe, and Ralph Wachter, editors, *Topology and Category Theory in Computer Science*, pages 357–390. Oxford, 1991.
17. Joseph Goguen. Ontology, society, and ontotheology. In Achille Varzi and Laure Vieu, editors, *Formal Ontology in Information Systems*, pages 95–103. IOS Press, 2004. Proceedings of FOIS'04, Torino, Italy.
18. Joseph Goguen. What is a concept? In Frithjof Dau and Marie-Laure Mungier, editors, *Proceedings, 13th Conference on Conceptual Structures*, volume 3596 of *Lecture Notes in Artificial Intelligence*, pages 52–77. Springer, 2005. Kassel, Germany.
19. Joseph Goguen. Information integration in instutions. In Lawrence Moss, editor, *Memorial volume for Jon Barwise*. Indiana, to appear.
20. Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, January 1992.
21. Joseph Goguen and Răzvan Diaconescu. Towards an algebraic semantics for the object paradigm. In Hartmut Ehrig and Fernando Orejas, editors, *Proceedings, Tenth Workshop on Abstract Data Types*, pages 1–29. Springer, 1994. Lecture Notes in Computer Science, Volume 785.
22. Joseph Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs.* MIT, 1996.
23. Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992. Drafts exist from as early as 1985.
24. Joseph Goguen and Grigore Roşu. Institution morphisms. *Formal Aspects of Computing*, 13:274–307, 2002.
25. Joseph Goguen and Grigore Roşu. Composing hidden information modules over inclusive institutions. In Olaf Owe, Soren Krogdahl, and Tom Lyche, editors, *From Object-Orientation to Formal Methods: Essays in Honor of Johan-Ole Dahl*, pages 96–123. Springer, 2004. Lecture Notes in Computer Science, Volume 2635.
26. Joseph Goguen, James Thatcher, and Eric Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In Raymond Yeh, editor, *Current Trends in Programming Methodology, IV*, pages 80–149. Prentice-Hall, 1978.
27. Joseph Goguen, James Thatcher, Eric Wagner, and Jesse Wright. Initial algebra semantics and continuous algebras. *Journal of the Association for Computing Machinery*, 24(1):68–95, January 1977.

28. Joseph Goguen and William Tracz. An implementation-oriented semantics for module composition. In Gary Leavens and Murali Sitaraman, editors, *Foundations of Component-based Systems*, pages 231–263. Cambridge, 2000.
29. Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*, pages 3–167. Kluwer, 2000.
30. Georg Gottlob. Computing cores for data exchange: new algorithms and practical solutions. In Georg Gottlob, editor, *Proc. Principles of Database Systems (PODS) 2005*, pages 148–159. ACM, 2005.
31. John Gray. Sheaves with values in a category. *Topology*, 3(1):1–18, 1965.
32. Richard Hull and Masatoshi Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In Dennis McLeod, Ron Sacks-Davis, and Hans-Joörg Schek, editors, *Proc. 16th Conference on Very Large Databases (VLDB'90)*, pages 455–468. Morgan Kaufman, 1990.
33. Yannis Kalfoglou and Marco Schorlemmer. Information-flow-based ontology mapping. In Robert Meersman and Zahir Tari, editors, *Proc. Intl. Conf. on Ontologies, DataBases, and Applications of Semantics for Large Scale Information Systems*, volume 2519 of *Lecture Notes in Computer Science*, pages 1132–1151. Springer, 2002.
34. Saunders Mac Lane. *Categories for the Working Mathematician.* Springer, 1998. Second Edition.
35. Till Mossakowski, Joseph Goguen, Razvan Diaconescu, and Andrzej Tarlecki. What is a logic? In Jean-Yves Beziau, editor, *Logica Universalis*, pages 113–133. Birkhauser, 2005. *Proceedings*, First World Conference on Universal Logic.
36. Young-Kwang Nam, Joseph Goguen, and Guilian Wang. A metadata integration assistant generator for heterogeneous distributed databases. In Robert Meersman and Zahir Tari, editors, *Proc. Intl. Conf. on Ontologies, DataBases, and Applications of Semantics for Large Scale Information Systems*, volume 2519 of *Lecture Notes in Computer Science*, pages 1332–1344. Springer, 2002.
37. Benjamin C. Pierce. *Basic Category Theory for Computer Scientists.* MIT, 1991.
38. Lucian Popa, Yannis Velegrakis, Renee Miller, and Mauricio Hernandez. Translating web data. In *Proc. Conference on Very Large Databases (VLDB'02)*, pages 598–609. VLDB, 2002.
39. Erhard Rahm and Philip Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.
40. Jérôme Siméon and Philip Wadler. The essence of XML. In *Proc. Principles of Programming Languages*, pages 1–13. ACM, 2003.
41. Andrzej Tarlecki, Rod Burstall, and Joseph Goguen. Some fundamental algebraic tools for the semantics of computation, part 3: Indexed categories. *Theoretical Computer Science*, 91:239–264, 1991.
42. Guilian Wang, Joseph Goguen, Young-Kwang Nam, and Kai Lin. Critical points for interactive schema matching. In Jeffrey Xu Yu, Xuemin Lin, Hongjun Lu, and YanChun Zhang, editors, *Advanced Web Technologies and Applications*, pages 654–664. Springer, 2004.