**Kinds**

* Inference by analogy. Model : Posible functional inferences: Datatypes, (upper) schema, instances (Sets / Kinds) domain / range.
* Dimensional (Dimension, Measure, Unit, Value). Unit: domain / range.
* Relationships (Relationship, Relation, Role, Entity). Role: domain / range.
* (Dimension, Event : Measure) (Functorial Inference Dimension / Relationship) (Dimension, Event : Measure);
* Sets / Models (transforms) / Relations (primitives: i.e. set members complements). Orders. Sets: discrete categories: schema types, domain / instances types (sets).
* Notation and terminology. For any natural number n ∈ N, let n denote the set {1, 2, . . . , n}. We sometimes regard sets as discrete categories without mentioning it. Note that 0 = ∅. Let [n] denote the linear order 0 ≤ 1 ≤ . . . ≤ n. We sometimes regard orders as categories without mentioning that either. In particular 1 is the terminal category; it has one object and one morphism (the identity). Given any category C, we denote the category of all functors C → Set by C–Set.
* The terminal object in C–Set sends each object in C to 1; we denote it by 1C : C → Set. For any category C, there is a one-to-one correspondence between the objects in C and the functors 1 → C. Thus we may denote an object c ∈ Ob(C) by a functor 1c−→ C. In particular, we elide the difference between a set and a functor 1 → Set. Custom datatypes: dimensional functions domain / range in operations / predicates (distance / time).
* Literals / Blank nodes identity: instances / types / values of an addresable measure.
* Rules / Inferences Alignments.
* Reactive Activation: Inferences. Model Rules: N3 / Turtle / DSL. Templates: Resources, Kinds, Contexts Encoding (roles) for Functional Reasoning (Predicates: schema / values)
* XML / XSLT DTDs / XSD. RDFS / OWL OGM (Templates Encoding).
* Alignment Inferences (Functional Predicates): sameAs, greaterThan, lessThan, equals, partOf, parentOf, siblingOf, previousOf, nextOf, roles (schema / values).
* Functional Composite inferred Predicates:
* greaterThan([a.age](http://a.age), [b.age](http://b.age)) : older(a, b) : Activation.
* Predicates: Templates. Resources, Kinds, Contexts Encoding (Function predicates argument mappings). Composite from primitives / roles (Contexts).
* [business.products.premium](http://business.products.premium)
* Inference (Functions same results) Ontology Matching.
* Order:
* Mappings from Resource Types hierarchy lattice.
* State order (in context class hierarchies axes), comparison relations, iterations, flow, events, causal relations, units, enums, equivalence, etc.
* Data order: Resource Kind hierarchies.
* Schema order: Role Class hierarchies.
* Encoding: Magic numbers. Resource Content Type Hash. ID Hashing: block (DIDs) result of inferences chain (event sourcing). Encoding: addresses.
* Model declared as Interaction Layer Augmentation(s) (matching Messages) in Interaction Model. Flows. Model: possible inferences (dataflow).
* AI for Understanding Human Goals
* "In the quest to capture ... social intelligence in machines, researchers from MIT’s Computer Science and Artificial Intelligence Laboratory (CSAIL) and the Department of Brain and Cognitive Sciences created an algorithm capable of inferring goals and plans, even when those plans might fail."
* "... ability to account for mistakes could be crucial for building machines that robustly infer and act in our interests ... Otherwise, AI systems might wrongly infer that, since we failed to achieve our higher-order goals, those goals weren’t desired after all. We’ve seen what happens when algorithms feed on our reflexive and unplanned usage of social media, leading us down paths of dependency and polarization. Ideally, the algorithms of the future will recognize our mistakes, bad habits, and irrationalities and help us avoid, rather than reinforce, them."
* <https://scitechdaily.com/new-mit-social-intelligence-algorithm-helps-build-machines-that-bette>
* ("Inference" is used broadly herein to mean any rule or procedure that produces new assertions from existing assertions -- not just conventional inference engines or rules languages.)
* Furthermore, applications often need to perform custom "inferences" (or data transformations) that are not convenient to express in available (non-standard) rules languages, such as RDF data transformations that are needed when merging data from independently developed sources having different data models and vocabularies.  And merging independently developed data is the \*most\* fundamental use case of the Semantic Web.
* One possibility for addressing this need might be to embed RDF in a full-fledged programming language, so that complex inference rules can be expressed using the full power and convenience of that programming language.  Another possibility might be to provide a convenient, standard way to bind custom inference rules to functions defined in a programming language. A third possibility might be to standardize a sufficiently powerful rules language.
* Here’s a JavaScript-based language for path queries, which reduce things such as “the user’s list of friends” to three words ([user.friends.label](http://user.friends.label)) instead of a SPARQL query:
* – <https://github.com/solid/query-ldflex>
* – <https://solid.github.io/ldflex-playground/>
* Custom Datatypes. Blank nodes. Dimensions. HyTime.
* isn't the structure for that already present in RDF by datatypes in the syntax and D-extensions in the semantics?
* So, it would just be a standardised way to define the lexical space of a datatype (ABNF or the like) plus something to define the operations and the semantics/value space, but there is no modification to RDF itself needed.
* Compare Defined Datatypes: Facets / Axis domain / content type. Type value / reference (context struct types / values)
* When I say that (1,2) is a true value, aka an immutable struct, your
* answer is that two (1,2) values are not the same because, taking into
* account the open world assumption, they could have a third dimension
* (or some other attribute).
* You write "the same literal value is assigned to two nodes, does not
* make them the same".  Is it correct to rephrase that as follows ?
* p1 has\_coords (1,2)
* p2 has\_coords (1,2)
* In that case I agree that nothing proves p1 and p2 are the same.
* But what I am pointing at when I talk about an immutable struct is not the above.
* A better comparison would be "2002-05-30T09:00:00"^xsd:datetime, that
* could be deserialized to (year: 2002, month: 5, day: 30, hour: 9,
* minutes: 0, seconds: 0).
* Would you say that the two literals
* "1;2"^<<http://mydomain.com/mytypes/tuple-of-two-integers>> and
* "1;2"^<<http://mydomain.com/mytypes/tuple-of-two-integers>> are
* different things ?
* Does it follow from the open world assumption that
* "2002-05-30"^xsd:date and "2002-05-30"^xsd:date are different values
* because one could append the time information and write
* "2002-05-30T09:00:00"^xsd:datetime ?
* I would think that the open world assumption applies to nodes, not to
* values/literals. Am I missing something ?
* Compare / Translation / Equivalence: Dimension. Data type.
* I just wish we had allowed datatypes which used more than one character string, so that (for just one example that caused way too much hassle) language-tagged strings, but also things like latitude+longitude or number+ unit (5 inches, 27 cm, 3.5 kg) could have been handled naturally. Right now it is not easy to say in RDF that the Thames is 215 miles long, and also that 215 miles is the very same thing as 346 km. But this kind of thing is ubiquitous.
* So maybe, rather than a literal or a bnode, RDF could just incorporate some JSON? Can put it all on one line like a literal or bnode, and can use nesting too.
* Example triples (I've removed string quotations etc. because this is just rough pseudocode):
* france name {type: LanguageTaggedString, value: France, language: English}
* place1 geoCoordinates {type: GeoCoordinates, latitude: 0.0, longitude: 0.0}
* thames length {type: QuantitativeValue, value: 346000, unitCode: MTR}
* uiElement shape {type: Circle, x: 0, y: 0, radius: 10
* laptop1 tempGT laptop2
* Inferences: RDFS, RDF\*, OWL, SPARQL, Turtle, N3, Trig, Shapes, Monads, Zippers. Reactive: inferences (functors) domain / range dataflow describes models.
* Static: Bus. Functor base predicates.
* Dynamic: Instances Functors predicates.
* Monads: reference / value types.
* this is related to hierarchical URIs:
* <http://patterns.dataincubator.org/book/hierarchical-uris.html>
* In your case, the question is how you have organized the collections/items of basic and admin persons in your dataset.
* One option is that both "basic persons" and "admin persons" belong to the same collection and have a single URI pattern: /persons/{id}
* In this case you cannot tell if resource /persons/12345 is a "basic person" or "admin person" just from its URI. You need to dereference it and the look into RDF types and properties. Another option is that you treat them as belonging to separate collections, for example: /persons/{id} and /admins/{id}
* In this case you can easily tell if a resource is a "basic person" or an "admin person" already from its URIs.
* Linked Data Templates are best suited for this second case, where URI space is subdivided into hierarchies based on entity types. That makes it easy to define URI templates that match precisely the set ofresources that you want
* Referrer Facets:
* Ah, I might have explained our case bit vaguely. So I just meant that we have in RDF data one kind of person resources, and depending on the access rights in the application, you are allowed to see different portions of that person's data.
* Basic user sees only the name, for example, and admin user is allowed to see all data. This is handled by selecting different template for basic user and admin, right?
* So as with your first example:
* /person/basic\_access/{id}
* --
* :BasicPersonAccessItem a ldt:Template ;     ldt:match "/person/basic\_access/{id}" ; ldt:query :ConstructBasicPerson ;
* ----
* /person/admin\_access/{id}
* --
* :AdminPersonAccessItem a ldt:Template ;     ldt:match "/person/admin\_access/{id}" ;
* ldt:query :ConstructFullPerson ;
* And this acl example
* /person/{agent}/{id}
* --
* :PersonAccessItem a ldt:Template ;     ldt:match "/person/{agent}/{id}" ; ldt:query :ConstructPerson ;
* Blank Nodes: Registry, Naming, Index, matching Augmentations:
* I think you'll end up with false negatives that way though. I think comparison operations for value types need to be type dependent.
* Example 1:
* id: \_:fraction1
* type: Fraction
* numerator: 1
* denominator: 2
* id: \_:fraction2
* type: Fraction
* numerator: 2
* denominator: 4
* The algorithm will return different IDs, but they're the same value.
* Example 2:
* id: \_:fraction1
* type: Fraction
* numerator: 1
* denominator: 2
* batteryPercentageOf: laptop1
* id: \_:fraction2
* type: Fraction
* numerator: 1
* denominator: 2
* batteryPercentageOf: laptop2
* Again, the algorithm will return different IDs, but they're the same value.
* Something that I think might assist in this area would be if mainstream value types had accompanying comparison operations.
* I agree regarding Example 1. In Example 2, I think that \_:fraction1 and
* \_:fraction2 are different things (they are readings for different
* laptops; I would not say, for example, that two people are the same
* because they share the same date of birth).
* I think the general problem you refer to resides at a different level
* and not really related to blank nodes. Note that if you use IRIs:
* id: :fraction1
* type: Fraction
* numerator: 1
* denominator: 2
* id: :fraction2
* type: Fraction
* numerator: 2
* denominator: 4
* You end up with the same issue of :fraction1 and :fraction2 being in some sense related, arguably owl:sameAs, but not being recognised as such "automatically". There is no way to resolve this at the RDF level, and nor, I believe, should there be, as it would over-encumber RDF.
* Someone, somewhere, has to either (1) define what makes two things "the same value", or (2) provide lots of examples of things that are "the same value" over which supervised learning can be applied (or (3) perhaps both). There is lots of machinery for (1) in OWL, for example, though your precise example would not be covered as arithmetic is limited.
* Example 2 is just a different shape of the following graph:
* laptop1
* batteryPercentage: \_:fraction1
* laptop2
* batteryPercentage: \_fraction2
* The algorithm is going to give false negatives with one shape but not the other, even though the meaning of the graphs is the same.
* It also has the potential to give false positives. It'll generate the same ID for both people in the following graph, even though they might not necessarily be the same person:
* \_:person1
* firstName: Mary
* lastName: Smith
* \_:person2
* firstName: Mary
* lastName: Smith
* I don't think there's an easy way around this sort of thing aside from type-dependent comparison algorithms. Swift developers are used to it actually, we have to define comparison algorithms when we define custom types: <https://developer.apple.com/documentation/swift/equatable>
* ...though we will too often end up with differing ids for nodes representing the same real world thing.
* The idea of taking knowledge of functional and inverse-functional properties into account is interesting (perhaps in some post-parsing canonicalization step aka "smushing"...).
* From my own narrow perspective, the single thing that would make RDF more successful would be universal adoption of labeled property graphs, RDFStar, SPARQLStar, a standardized CSV/TSV format for semantic LPGs, and an alternative OWL layering (see <https://douroucouli.wordpress.com/2019/07/11/proposed-strategy-for-semantics-in-rdf-and-property-graphs/> and <https://github.com/cmungall/owlstar>). This level of abstraction would hide/eliminate most of the blank nodes I see, and would give people the level of abstraction they really want for modeling, and would match up with the tools and databases people use outside our semantic web bubble.
* If a (possibly composite) key is known for an object, then other properties can and should be ignored in computing a canonical node name for the object, so that some degree of automatic graph leaning can occur, which would be quite helpful.
* In fact, I've started to think that \*every\* object should be required to have a (possibly composite) key, just like with standard relational database practice.  A higher-level RDF-ish syntax could even enforce such a rule.
* Literals seem like syntactic sugar for blank nodes, so whatever applies to blank nodes seems like it should apply to literals too, for example:
* "2020-01-01T00:00:00"^^xsd:dateTime
* The literal seems to only exist as a convenient way of writing:
* \_:dateTime1
* type: xsd:dateTime
* year: 2020
* month: 01
* day: 01
* hour: 00
* minute: 00
* second: 00
* In Swift, you write code the normal way using Int and Float etc., but behind the scenes and hidden from users they're actually implemented as structures. So one could alternatively argue that literals must die (joking).
* I really appreciate all the replies, thank you.
* Why not use the literal?
* I'm at the edge of my RDF knowledge here, but say I want to define my own composite value type, like Circle for example?
* The way I'd imagine doing it is:
* Circle
* type: rdfs:Datatype
* \_:circle1
* type: Circle
* center: \_:coordinate1
* radius: 10
* In this specific case it could be rdf:type time:DateTimeDescription from OWL-Time. See <https://www.w3.org/TR/owl-time/#time-position>
* That's right, Simon. Correct me if I'm wrong though, using
* type: time:DateTimeDescription
* versus:
* type: xsd:DateTime
* makes one a reference type and the other a value type.
* The "Description" suffix leads to a little confusion I think. By the same logic xsd:DateTime could be named xsd:DateTimeDescription, I think time:DateTime might have been sufficient. The Circle example might be a better example in any case.
* You are at the edge of my knowledge of what you are wanting to do. The RDF specs deliberately did not restrict datatypes to any particular collection, with the intention and expectation that people and organizations would define new datatypes. So if you want to have a ‘circle’ datatype, then go ahead a define it. It would be helpful to define it as exactly as you can, and of course it needs to conform to the basic rules of RDF datatypes, but that should not be too hard. If you publish a document at the URL of the datatype - say at https://www/moretti/mydatatype/Circle explaining what it means, that would be especially helpful. Then just write the literal “10”^^https://www/moretti/mydatatype/Circle to refer to it in RDF. Now, admittedly, this will only be fully ‘understood’ by RDF engines which know about your datatype, but others are required to treat it just as they would treat an unknown URI, so nothing should break. And in this case of course you can properly assert https://www/more
* What I have seen time and again in modeling is that you start out thinking you can use a simple literal value, but later you find it needs to become a composite.  Say you are modeling a project.  It has a budget, and that's just a number, a literal value.  But soon you need it to have a capital improvement component and a maintenance component, and you find you also need it split out into quarterly segments.  Your nice simple literal has become a complicated construction.
* So if I'm able to use rdfs:Datatype in that way, then during processing, blank nodes whose types are instances of rdfs:Class should be given a URI (using UUIDs for example), but blank nodes whose type is an instance (singular) of rdfs:Datatype shouldn'[t.The](http://t.the) second:If I'm able to do that, then literal syntax only exists as syntactic sugar for blank nodes whose type is an instance of rdfs:Datatype. So if all blank nodes should become a subset of the graffiti of S then all literals should [too.The](http://too.the) whole issue is interesting to me because since 2015 iOS developers have been forced to understand value semantics with the introduction of structs in Swift, and I see parallels in how the RDF community continuously struggles with blank nodes. I feel like rdfs:Datatype could have a bigger role to play than it currently does and potentially help clarify things. But I'm no logician, it's just an intuition of mine that could be very very very wrong, I don't know
* No, I mean blank nodes whose type is an instance of rdfs:Datatype, so a Circle value for example, or a xsd:DateTime value described without using literal syntax.
* Basically what I'm saying is if in Swift you can distinguish:
* class Circle {
* center: coordinate1,
* radius: 10,
* }
* struct Circle {
* center: coordinate1,
* radius: 10,
* }
* In RDF I feel it should be something similar like:
* circle1
* type: Circle (class)
* center: coordinate1
* radius: 10
* \_:circle1
* type: Circle (datatype)
* center: coordinate1
* radius: 10
* But it's not, what you actually have to do is something like:
* circle1
* type: Circle (class)
* center: coordinate1
* radius: 10
* "X0Y0R10"^^Circle
* It seems like something is missing, and that literals are a stand-in for that missing something.
* The latter can be seen as a value, like a number or a character
* string, but a composite one. Think of the position of a point with
* coordinates x,y.
* Precisely, Nicolas, a composite value.
* In a sense, the form "1;2" is a serialization of the Point
* dataclass.
* Yes, very well put.
* Besides Circle and Coordinate, any of <https://schema.org/StructuredValue> and its subtypes could also suitably be described as type: rdfs:Datatype and ideally not require a literal syntax each.
* In the example Anthony gives (that I think I understand well because I
* have done a lot of programming and teaching of programming), there is
* a class, with members that can change, and a datastructure, with
* attributes that do not change (that some programming languages call a
* dataclass).
* The latter can be seen as a value, like a number or a character string, but a composite one. Think of the position of a point with coordinates x,y.
* IIUC, Anthony is pointing us at: I could use a Literal "1;2"^^Point, but it would be nicer to have some way to express that "1" and "2" are numbers and that there is no difference between the point at coordinates (1,2) and the point at coordinates (1,2), in the same way that there is no difference between the literal "3.14" and the literal "3.14". In a sense, the form "1;2" is a serialization of the Point dataclass.

A Proposal for the Characterization of Multi-Dimensional Inter-relationships of RDF Graphs Based on Set Theoretic Approach. Sets, Resources CSPO Roles, Kinds, Contexts Inter Graph Traversal.

Subject-Subject and Predicate-Predicate relationship

Subject-Subject and Predicate-Predicate relationship characterizes the

specific criteria of RDF Graph relationship, where two RDF Graphs T1 and T2

share common subject and predicate. The significance of these criteria is that

two statements are semantically equivalent from the Subject and its property

perspective. The only difference exists in a point that the two statements have

different values for the same properties of the same subjects. It is evident that

this criterion dictates a strong relationship between two RDF Graphs T1 and

T2 and between two corresponding statements as well.

Subject-Subject and Predicate-Predicate relationship between two RDF

Graphs T1 and T2, if the following set theoretical expressions are all true.

Sub(T1) int Sub(T2) not empty;

Obj(T1) int Obj(T2) not empty;

Sub(T1) int Obj(T2) not empty;

Obj(T1) int Sub(T2) not empty;

E1 int E2 not empty;

Object-Object and Predicate-Predicate relationship

Object-Object and Predicate-Predicate relationship identifies the criteria of

RDF Graph relationship, where two RDF Graphs T1 and T2 share common

object and predicate. The significance of this criterion can be exhibited in

those cases where two statements are semantically equivalent from the

property and its value perspectives. Two RDF Graphs related with this kind of

condition, must have different subjects which hold same property with same

values. Conditions for Object-Object and Predicate-Predicate relationships

are presented in Fig.2. using Venn diagram schema.

Mathematically, there exists a Object-Object and Predicate-Predicate

relationship between two RDF Graphs T1 and T2, if the following set

theoretical expressions are all true.

Obj(T1) int Obj(T2) not empty;

Sub(T1) int Sub(T2) not empty;

Sub(T1) int Obj(T2) not empty;

Obj(T1) int Sub(T2) not empty;

E1 int E2 not empty;

Subject-Predicate relationship

Subject-Predicate relationship has a different significance and consequence

than the other two types of relationships discussed above. In this case, two

RDF Graphs T1 and T2 never share their subject, object or predicate, rather

the resource described by one's subject is same as that of resource described

as predicate of others. With this condition, the subject of one statement acts

as a property of the other statement. The two RDF Graphs related with their

Subject - Predicate relation can represent complex indirect search construct.

The two statements with completely different subjects could be linked with

each other through this relationship.

Mathematically, there exists a Subject-Predicate relationship between two

RDF Graphs T1 and T2, if the following set theoretical expressions are all

true.

Sub(T1) int Sub(T2) not empty;

Obj(T1) int Obj(T2) not empty;

Sub(T1) int E2 not empty;

E1 int E2 not empty;

DATABASE QUERIES AND CONSTRAINTS VIA LIFTING

PROBLEMS (Models : Inferences)

Abstract. Previous work has demonstrated that categories are useful and expressive models for databases. In the present paper we build on that model, showing that certain queries and constraints correspond to lifting problems, as found in modern approaches to algebraic topology. In our formulation, each so-called SPARQL graph pattern query corresponds to a category-theoretic lifting problem, whereby the set of solutions to the query is precisely the set of lifts. We interpret constraints within the same formalism and then investigate some basic properties of queries and constraints. In particular, to any database π we can associate a certain derived database Qry(π) of queries on π. As an application, we explain how giving users access to certain parts of Qry(π), rather than direct access to π, improves ones ability to manage the impact of schema evolution.

Main example of a lifting query. We now provide an example of a situation

in which one may wish to query a database, and we show that this query naturally takes the structure of a lifting problem. We break a single example into three parts for clarity.

Example 1.1.1 (Main Example 1: Situation, SPARQL, and schema). Suppose you have just come home from a party. There, you met and really hit it off with a married couple; the husband’s name is Bob and the wife’s name is Sue; they live in Cambridge. From your conversation, you know that Bob works at MIT and Sue works in the financial sector. You’d like to see them again, but you somehow forgot to ask for their contact information; in particular you’d like to know their last names. This is a typical database query problem. It can be phrased as the following SPARQL graph pattern query (which we arrange in two columns for space and readability reasons):

(4)

(?marriage includesAsHusband ?b) (?marriage includesAsWife ?s)

(?b hasFirstName Bob) (?s hasFirstName Sue)

(?b livesIn Cambridge) (?s livesIn Cambridge)

(?employedb is ?b) (?employeds is ?s)

(?employedb hasEmployer MIT) (?employeds hasEmployer ?sueEmp)

(?sueEmp isIn financial)

(?b hasLastName ?bobLast) (?s hasLastName ?sueLast)

The query in (4) might be asked on the following database schema:

Given that S is instantiated with data π : I → S, one can hope to find Bob and Sue, and then determine their last name. In the following two examples (Examples 1.1.2 and 1.1.3) we will show that this query corresponds to a lifting problem for π.

Example 1.1.2 (Main Example 2: WHERE-clause and Result schema). Recall the SPARQL query presented as (4) in Example 1.1.1, in which we wanted to find information about our new friends Bob and Sue. We will use a lifting problem to state this query; to do so we need to come up with a result schema R, a constraint schema (a set of knowns) W, and a mapping m: W → R embedding the known objects into the result schema. In this example we will present m, W, and R. In Example 1.1.3 we will explain the lifting diagram for the query and show the results.

In order to find our friends Bob and Sue, we will use the following mapping:

W:=

(Y1:MIT)

(T:financial sector)

(F1:Bob)

(C1:Cambridge)

(C2:Cambridge)

(F2:Sue)

> m >

R:=

(Y1:an employer)

(G:a marriage) includes as husband, includes as wife

(T:a sector)

(Y2:an employer) is in o

(E1:an employed person) is / has

(OO)

(P1:a person) has, lives in

(P2:a person) has, has, lives in

(E2:an employed person) is, has

(OO)

(F1) a first name

(L1) a last name

(C1) a city

(C2) a city

(L2) a last name

(F2) a first name

The functor m: W → R is indicated by sending each object in W to the object with the same label in R; e.g. pMITq in Ob(W) is sent to pan employerq in Ob(R) because they are both labeled Y 1.

To orient oneself, we suggest the following. Count the number of constants

in the SPARQL query (4)—there are 6 (such as Bob, Cambridge, etc.); this is precisely the number of objects in W. Count the combined number of constants and variables in the SPARQL query—there are 14 (there are 8 variables, such as ?marriage, ?empoyedb, etc.); this is precisely the number of objects in R. Finally, count the number of triples in the SPARQL query – there are 13; this is precisely the number of arrows in R. These facts are not coincidences.

Example 1.1.3 (Main Example 3: Lifting diagram and result set). In Example 1.1.2 we showed a functor m: W → R corresponding to the SPARQL query stated in (4). In this example we will explain how this query can be formulated as a lifting problem of the form which serves to pose our query to the database instance π. At this point we can ask for the set of solutions `. So far, W, m, R, and S have been presented, I and π have been assumed, and the set of `’s is coming later, so it suffices to present p

and n.

One should refer to our presentation of S in Example 1.1.1 (5). The functor

n: R → S should be obvious from our labeling system (for example, the object

E1=pan employed personq in category R is mapped to the object E=pan employed

personq in category S). Note that, as applied to objects, n is neither injective nor

surjective in this case: n −1 (P) = {P1, P2} and n −1 (D) = ∅.

Suppose π : I → S is our data bundle, and assume that it contains enough data

that the constants in the query have unique referents. 3 There is an obvious functor p: W → I that sends each object in category W to its referent in I. For example, we assume that there is an object in I labelled pMITq, which is mapped to by the object Y1=pMITq in W.

Thus our query from (4) is finally in the form of a lifting problem as in (6). We

will show in Example 4.3.4, after we have built up the requisite theory, that the

set of lifts can be collected into a single table, the most useful projection of which

would look something like this:

Marriage

ID Husband Wife

ID First Last City ID First Last City

G3801 M881-36 Bob Graf Cambridge W913-55 Sue Graf Cambridge

This concludes the tour of our main example: we have shown a typical query

formulated as a lifting problem. The mathematical basis for the above ideas will

be presented in Section 4.

Introduction

2. Elementary theory of categorical databases 9

3. Constraints via lifting conditions 15

4. Queries as lifting problems 25

5. The category of queries on a database 34

6. Future work

Purpose of the paper. The purpose of this paper is to:

• provide an efficient mathematical formulation of common database queries (modeling both SQL and SPARQL styles),

• attach a geometric image to database queries that can be useful in conceptualization, and

• explore theory and applications of the derived database schema Qry(π) of queries on a database instance π, and the derived instance of results.

We include several mathematical results that are well-known to experts, for the purpose of aiding those interested in using this paper to bridge the gap between database theory and category theory.

“Queries on a database”. In wide-spread terminology for database queries,

a query cannot depend on the current instance π of the database, but instead only on the schema S. This is perfectly reasonable for theoretical and practical reasons. Often in applications, however, one uses what is known as a cursor, which is basically a pre-defined query consisting of a join-graph and a set of variables to be bound at run-time. With respect to the diagram the join-graph is R, the set of variables waiting to be bound is W, and the binding itself is p: W → I. The mathematics will be covered more extensively in Section 2; in the remaining paragraphs of Section 1.5.1, we hope to get across how one might connect our use of the term “query” in the present paper to common ideas in database systems.

|  |  |  |
| --- | --- | --- |
| W | p > | I |
| m √ | l | π √ |
| R | n > | S |

In applications, a query wizard may run the cursor in a 2-step query process: first it will query the database to offer the user a drop-down menu of choices in the active domain of each variable. The user will choose a row to which the variable will be bound (once for each variable). At this point the program will apply the actual query declared by the cursor. This two step process corresponds to searching for possible functors p: W → I and then searching for lifts.

Throughout this article, when we speak of queries on a database, we mean queries for which the constant variables have been bound to elements in the active domain of a given instance. However, as we will see in Section 4.2, one can also use the same machinery in cursor-like fashion to pose queries in which variable values have been chosen without regard for whether or not they are in the active domain. In other words, we will see that what can be accomplished by queries in the sense of traditional relational database theory fits easily into our framework. Because it works either way, we thought that the unusual terminology “queries on a database” would be best because it neither lulls the reader into thinking that these gadgets are completely instance-independent, nor frightens the reader into thinking that the instance must be known in advance for the ideas here to work.

In Section 3 we define constraints on a database in terms of lifting conditions and discuss some constraint implications. We give several examples to show how various common existence and uniqueness constraints (such as the constraint that a given foreign key column is surjective) can be framed in the language of lifting conditions. In Section 4 we discuss queries as lifting problems, and review the paper’s main example. In Section 5, we show that the information in a given database instance can be collected into a new, derived database. This derived database of queries and their results can be queried, giving rise to nested queries. We explain how this formulation can be useful for managing the impact of schema evolution. Finally in Section 6 we briefly discuss some possible directions for future work, including tying in to Homotopy Type Theory (in the sense of [Awo] and [Voe]) and other projects.