* Encodings:
* FCA Inferences. ISO TMRM / TMDM / TMCL / TMQL / HyTime.
* APIs: RDF / RDFS / RDF\*  OWL / OWL\* / SPARQL / SHACL / ShEx / Inferences / Formats / Syntaxes (inferences/augmentation/resolution models encoding: URNs lattice):
* FCA Predicates Contexts (S, p, O). P(S, O). Schema defined models. Verbs definitions: domain / range dataflow.
* FCA Scaling: Resources, Kinds, Contexts Functional Categories levels. Reification(\*). Transform. Traversal.
* FCA DCI: Metaclass, Class, Instance, Context, Role, roles Occurrences scaling lattice
* FCA URNs: See Sets Model (Resources, Kinds, Contexts) Functional API:
* schema:[context]/[ctx:subject]/[ctx:predicate]/[ctx:object]#[hashResource]?[queryString]
* [bodyResource] : HTTP Headers. Content-type, content / body. Encode Graph from navigation / traversal metadata.
* [ctx:resource] : Lattice Context plus concept occurrence.
* FCA: Contexts(Object, Attribute); Nested Contexts. Scaling.
* Resources: S(Predicate, Object);
* Resources: P(Subject, Object);
* Resources: P(Object, Subject);
* Kinds: Scaling (aggregates resources).
* S(P, O); SK.
* P(S, O); PK.
* P(O, S); OK.
* Contexts: Scaling. Aggregates Kinds.
* SK(PK, OK); Statement.
* PK(SK : S, OK : O); Mapping.
* PK(OK : O, SK : S); Transform.
* 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);
* Bookmarks / Bibliography / Tools.
* Domains:
  + BI. Pentaho.
  + ERP. Tryton.
  + DDD: Instant API Backends
  + Social / Purposes: Solid / StratML.
* Features (RDF4J Sails):
  + Components:
  + Connectors: Traits. CDI Bus Signatures / Protocols (Events Encoding)
  + MDM: Onto Merge Matching. Traceability / Graphs Traversal. Models Bus.
  + ESB: Integration: Connectors Bus.
  + Rules / Inferences.
  + Workflows.
  + TMDM, TMRM: ISO TopicMaps.
  + FCA: Formal Concept Analysis.
  + Sails stack: from plain RDF / RDFS / OWL / Sem Web stack inferences through Augmentation Sail(s) to DDD Runtime: OGM / DCI HATEOAS Applications.
  + Augmentation Sail:
  + Alignment: Naming. Resources Model. Available Interactions Data.
  + Aggregation: Registry. Controller. Sets Model. Available Contexts Interactions. Dataflow.
  + Activation: Index. View. Key Valuel. Available Data Contexts.
  + Naming: Resources Model. Model. Available Interactions Data.
  + Registry: Queries. Sets Model. Controller. Available Contexts Interactions.
  + Index: Hierarchical Key Value (FCA / TMRM) store. Events Sourcing. View. Available Data Contexts.
  + Encoding / Resolution: FCA / TMRM Concept Lattice (nested key value bitstrings). Event Sourcing: Lattice order relation.
  + Dataflow:
  + Input:
  + Encode / Match Resource Data.
  + Browse / Augment Index for Resource (Data) Available Contexts.
  + Query / Augment Registry for Available Context Data Interactions.
  + Match / Augment Naming for Context Data Interaction Data.
  + Output:
  + Input Steps Dataflow resolve streams aggregated in a fan in / fan out fashion. Functional schema / domain transform / mappings / inferences applied.
  + Browser Extensions. Clients Connectors.
  + Deployment Connectors: Google Apps. Solid. DIDs.
  + Runtime: OGM / DCI: OpenRDF Elmo. Bus Endpoints. DOM HATEOAS.
  + Runtime: Qi4j (RDF Entity backend). Sesame. Bus Endpoints. DOM HATEOAS.

**Encodings:**

FCA Inferences. ISO TMRM / TMDM / TMCL / TMQL / HyTime.

APIs: RDF / RDFS / RDF\* OWL / OWL\* / SPARQL / SHACL / ShEx / Inferences / Formats / Syntaxes (inferences/augmentation/resolution models encoding: urns lattice):

FCA Inferences. ISO TMRM / TMDM / TMCL / TMQL / HyTime.

APIs: RDF / RDFS / RDF\* OWL / OWL\* / SPARQL / SHACL / ShEx / Inferences / Formats / Syntaxes (inferences/augmentation/resolution models encoding: URNs lattice):

FCA Predicates Contexts (S, p, O). P(S, O). Schema defined models. Verbs definitions: domain / range dataflow.

FCA Scaling: Resources, Kinds, Contexts Functional Categories levels. Reification(\*). Transform. Traversal.

FCA DCI: Metaclass, Class, Instance, Context, Role, Occurrence.

FCA URNs:

schema:[predicateContext]/[subjectResource]/[predicateResource]/[objectResource]#/[hashResource]?]/[queryString]

[bodyResource] : HTTP Headers. Content-type, content / body. Encode Graph from navigation / traversal metadata.

* 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.

**Workflows (Domain Goals) framework:**

Semantic Distributed Backends StratML rendered into vitualized applications environment (collaborative dashboards, wizards, guided flows).

Workflow kind: Need, Good, Product. Roles. Inter-domain workflows.

Prompts: context wiki (domain learning roles, domain evaluations: skills).

Prompts: Resource matching domain skills answers flows: values / decisions.

Prompt: value / decision query. User / Service response.

Workflow tray: declarative API interface (workflow concepts). Guided assistant / learning.

App: questions with a purpose. Domains roles / skills filters.

Workflows / Prompts Model Relations / Services:

Domains: Data, Information, Knowledge.

Products / Goods / Needs.

Goals: Use Cases. Data, Contexts, Interactions.

Profiles.

Roles.

Organizations (ad hoc).

Workflows / Prompts Model Relations / Services:

(Domain, Flow : Flow, Role : Relationship, Item : Entity);

Domain<Flow[]>;

Flow: Use Case. Produced / available for Domains.

Role: Prompts. Produces / available in state of Flow. Item: product, prompts consumer / producer kind.

Item: Product / Good / Need (Goal). Produces / populated by Roles (Prompts).

(DomainService : Relation, Resource : Domain, Context : Flow, Role : Relationship);

Retrieves available workflows Roles for Flows in Domains (ActivationService).

**Meta Meta Model:**

DOM OntResource: Models Meta Meta Model. Reference Model. Runtime / dataflow APIs.

Parsing: AST Object URL monad layers hierarchy. Parser combinators. Monad Zippers (tree encoding). Order. Flows (streams). Parser combinators.

OntResource:

Reactive (producer / consumer) entity. Hierarchy templates determines signatures / dataflows (topics / queues).

Signatures: observer (topics): ObjectID events stream. URL events stream. Role events stream. Layers hierarchy events set (reified: Resource, Statement, Kind, Relation events). Filter. Template / abstract methods. Core OntResource API behaviors. Dataflow.

Signatures: observer (topics): ObjectID events stream. URL events stream. Role events stream. Layers hierarchy events set (reified: Resource, Statement, Kind, Relation events). Filter. Template / abstract methods. Core OntResource API behaviors. Dataflow.

OntResource abstract APIs:

Abstract: Meta Model objects occurrences (aggregated by Object URL and Roles):

OntResource::ObjectID;

OntResource::Occurrences<URL, Role> : OccurrenceID;

OntResource represents aligned Objects / Resources of which the same Object (ObjectID) occurs in different Roles and (possibly) diferent URLs. Matching and alignments by reifying layers into abstract resources / OntResources.

Meta Model(s) APIs (Augmentation, Dataflow, Dimensional) reflects reference model events reacting to and producing signature matching messages on Meta Model(s).

OntResource's Roles are the DOM (Dynamic Object Model) types of the Meta Model(s) layers Occurrences / Objects / Contexts / SPOs. FCA Lattice contexts occurrences metadata. DOM APIs (reifying hierarchy layers, abstract templates):

Role: occurrence / object in CSPO slots. Denotes resource types in positions in statements (i.e.: Kind in Relation). Role CSPO is object / occurrence in statement occurrence position, Role type (i.e.: Kind, Relation) stated as Role instances in Meta Models with corresponding Kinds for its complimentary CSPO resources.

(Instance, Class, Metaclass, Occurrence);

(Context, Occurrence, Attribute, Value);

Meta Model(s). Augmentation APIs hierarchy:

Object<URL> : OntResource;

Abstract. OntResource aligned URL / template method design pattern.

Object<URL>;

Abstract. ObjectID.

Occurrence : Object, Object<URL> : OntResource;

Abstract ObjectID, OccurrenceID. Reify aggregated hierarchies.

Role : Occurrence, Occurrence, Object<URL>;

(Resource : Role, Role, Occurrence, Object<URL>);

(Kind : Resource, Resource, Role, Occurrence);

(Statement : Kind, Kind, Resource, Role);

(Relation : Statement, Statement, Kind, Resource);

Class hierarchy APIs:

(Instance, Class, Metaclass, Occurrence);

(Context, Occurrence, Attribute, Value);

**Meta Model(s)**

Dataflow APIs hierarchy:

Roles reify to Meta Meta Model Roles:

(Relation, Statement, Kind, Resource);

(Augmentation, Mapping / Predicate, Transform / Template, Resource);

Dimensional APIs hierarchy:

Roles reify to Meta Meta Model Roles:

(Relation, Statement, Kind, Resource);

(Dimension, Unit, Measure, Value);

Semiotic APIs hierarchy:

Roles reify to Meta Meta Model Roles:

(Relation, Statement, Kind, Resource);

(Context, Sign, Object, Value);

Domain Models. Service APIs. Templates.

Dataflow, Dimensional and Domain Models declaratively stated / reified in root Meta Meta Model Roles.

Messages. Encoding / I/O:

See 'Service Resources'.

(Class, Instance, Member, Value);

Augmentation:

Domain (Templates)

Meta Model (Activation)

Meta Meta Model (Augmentations)

Meta Model (Activation)

Domain (Templates)

Encodings:

(a (b (c (d, nil) : First / Rest binary tree.

(K: (K: inst, V: cls), V: (K: mcls, V: occur))

Object / Occurrence: Reify layers Object / Occurrence keys / values. Reference Model along with OntResource.

Message I/O (see Service Resources):

(Class, Inst, Attr, Val);

Templates: I/O Dataflow (aggregated messages) declarative bindings. Aggregated message facade (Augmentation, render / apply layer entity schema / data). Forms / Flows.

Forms / Flows: HATEOAS HAL. MVC. REST. Meta Models DCI based protocol.

HATEOAS: Forms / Flows Operations / Dataflow Representation / State IO (CRUD) prototypes / templates. Dialog. Prompts. Gestures. Context: navigation state (i.e.: pick operation value prompt shows value type Forms). DDD DOM.

Models Sets Reification (Populate DOMs / FCA):

OntResource

Object

Occurrence

Role : CSPO hiers Sets.

Resource : Role Set member.

Statement : Set members Role aggregation.

Kind: Statement Resources aggregations. Roles intersection sets.

Relation: Kind Statements aggregations. Transform: Kind Resources related to themselves (ID), then Relations to other Resource via Dataflow Kinds domain / range relationship (ordered).

Relation: aggregated aligned entities. Views (transforms). Kind members occurring in Statement Resource(s). Functors / Monads:

Relationship<A : Relation>::flatMap(F : Function<A : Relation, B : Relation>) : Relationship<B : Relation>;

Entity<A : Relationship>::flatMap(F : Function<A : Relationship, B : Relationship>) : Entity<B : Relationship>;

Function: declarative dataflow transform.

Dataflow Kinds domain / range: Grammar. Reify Kinds as CSPO and assert Statement. Aggregate further Kinds (until primitives).

Valid Statement (Grammar / Relation): domain / range, CSPOs backing assertions apply.

I/O: Parse / aggregate input Statements into corresponding Roles / Resources. Aggregate / match Kinds. Relations: render / activate. Resolve output Statements.

Built in Relation(s): ID, equals, inverseOf, parent, child, previous, next, etc. (upper ontology / meta model). Composites: Monad Zippers.

Meta Meta Mode (TBD):

DOM: OntResource, Object, Occurrence, Role. Express Meta Model levels metaclass, superclass, class, instance, occurrence relationships.

(Superclass, Class, Occurrence, Instance);

Metaclass: Role.

Occurrence: Role instance.

**Monadic Functors / Transforms:**

Parsed from / rendered to (materialized) to Meta Meta Model DOM Relations Monad Statements. Inheritance from Relation, Entity, Relationship, Flow.

Domain Models Entities / Relationships / Flows: Model parsed / rendered Relation materialized domain functors / transforms. Monadic layer for functional operation of underlying DOM Model Relations.

Model Relation:

(Relation, Statement, Kind, Resource);

ToDo: Resource selector API. (Relation).

ToDo: Transforms data flow. Application activation (available operations / state signatures / event flows). Browseable DOM / reified transforms (HATEOAS), reactive / event driven dialog / protocol: render applicable transforms.

Monad Transforms (hierarchically implemented):

Type Constructor.

Unit.

Bind.

Map / FlatMap.

CRUD (parametetized flatMap):

Resource::Append(Resource arg) : Function<Resource, Resource>;

Resource::Retrieve(Resource arg) : Function<Resource, Resource>;

Resource::Replace(Resource arg1, Resource arg2) : Function<Resource, Resource>;

Resource::Remove(Resource arg) : Function<Resource, Resource>;

Relations / comparisons:

Resource::Equals(Resource) : Function<Resource, Resource>;

Resource::Contains(Resource arg) : Function<Resource, Resource>;

Others: Primitives, Semiotic, Dimensional, etc. Domains modelled.

Domain Transforms:

Resource : Function (Mapping):

(Resource, Occurrence, Attribute, Value);

ActivationService:

Activation: Layers hierarchy (polymorphism), Layers reification.

Browse Layers contexts for available state transforms. Activation Resource states applicable to next / previous states / productions.

ActivationService Resource API:

Resource::Activate(ctx : Resource) : Resource[];

Resource::Available(Resource) : Resource;

Resource::Apply(Resource) : Function<Resource, Resource>;

DOM Layers Monads:

Resource<OntResource[]>; Aligned Resources (metaclass, class, instance, role, occurrence);

Kind<Resource[]> : Aligned Types.

Statement<Kind[]> : Aligned Statements.

Relation<Statement[]> : Aligned Assertions.

Entity<Relation[]>;

Relationship<Entity[]>;

Flow<Relationship[]>;

Domain<Flow[]>;

Entity Monad. Aligned Resource (data):

Entity<Relation[]>;

Entity<Relation[]> Monad. Relation[]: selector, Relations which are instances of / wrapped by this Entity scope.

Example: Employment anEmployment Entity. anEmployment Relation Statements. Selector matches Employment Relations.

Entity Monad Transforms:

Apply Relation selector / Relation CRUD to Employment anEmployment.

Relationship Monad. Aligned schema:

Relationship<Entity[]>;

Relationship<Entity[]> Monad. Entity[]: selector, Entities which are instances of / wrapped by this Relationship scope.

Example: Employmentship aggregation / concept of Employment Entities. Selector matches Entities Relations.

Relationship Monad Transforms:

Apply Entity selector / CRUD to Employmentship Employments.

Flow Monad. Aligned behavior:

Flow<Relationship[]>;

Flow<Relationship[]> Monad. Relationship[]: selector, Relationships which are instances of / wrapped by this Flow scope.

Example: SalaryRaised Flow for Employmentsip Employments. Selector matches Relationship Entities Relations.

Flow Monad Transforms:

Apply Employmentship selector / CRUD to SalaryRaised Employments.

Domain Monad. Aligned behavior:

Domain<Flow[]>;

**Backend Architecture (To Do):**

XML: XSL, XPath, XLink, XPointer, XQuery, XForms: hypermedia addressing / state / flows encoding / Message endpoints protocol.

Models.

Messages.

Encoding.

Endpoints.

Protocol.

Domain Connectors.

APIs.

Models:

Dispatch to Model layers context resources streams. Message IO. Endpoint Message matches in Model context: activation (Dataflow).

Messages:

Meta Meta Model entities (Relation). Meta Model layers scoped context statements.

Encoding:

Model layers scoped context statements.

Endpoints:

Model layers context resources streams (pub / sub topics). Topics: OntResources(s). Signatures: Dataflow Message IO wiring.

Protocol:

Message Dialog: i.e.: XML encoded Context statements Message IO with Model layer scoped prompts, placeholders, wildcards, variables. Models. Messages. Streams.

Domain Connectors:

Tools. Service Resource: URL, streams (Messages I/O).

Tools:

NakedObjects / Apache Isis. Apache MetaModel. JBoss Teiid. JDBC. JCA. Apache Stanbol. Apache Clerezza. OData / OpenAPI. JSON-LD. Spring HATEOAS / HAL. Apache Any23. D2RQ. R2RML.

Message Matching:

FCA Augmented Models: Context objects / attributes: layer quad resources Role, Kind, Resource.

Resource Context Concept: Resource x Kind.

Resource Context Object: Role x Resource.

Model / Schema Matching:

FCA Resource Context Concept.

Data / Resource Matching:

FCA Resource Context Object.

Matching Productions:

Result Set (query / augmentation result).

Augmentation (aggregation of new statements, alignment of new knowledge, activation of transforms / flows: result set).

Relationship / Entity monads mappings results.

FCA Scaling: Role > Kind > Resource aggregation of matching objects / attributes.

FCA Augment Resources: Role, Occurrence, Object (Resource), Concept, Object, Kind. Grammars. Match schema, instances. Mapping transforms: match behaviors.

FCA Contexts from Sets aggregation:

Set Roles: Context, Subject, Predicate, Object.

Sets aggregation: Statement, Kind (SuperKind, Kind, Attribute, Value) Attributes: class / Values: metaclass, Resource (Meta Model Roles: Kind context, Resource SPO), Context (Relation). Reified Kinds.

Aggregation streams: Sets reactive events aggregation. Sets (ordered) description APIs.

Sets aggregation: FCA Contexts scaled objects / attributes from Sets aggregation. FCAAPI.

**Dataflow. Components:**

Uniform Resource API: Sets, FCA, DOM layers, Monads. Reactive message driven dataflow (topics / signatures).

Inputs. Connectors / Services (active Resource topics).

Sets aggregation.

FCA Scaling. FCA Contexts (layers / occurrences).

DOM Layers / OntResource hierarchy. Augmentation, alignment, activation, matching. FCA alignments (concepts).

(Sets aggregation populates DOM layers FCA augmented or Sets aggregation builds FCA contexts rendered into FCA augmented DOM layers).

Functors. Parse DOM: Instantiate Relationship / Entity Monads (selectors / contexts). Model services interactions renders functors possible transforms as browseable (HATEOAS reified) resources / contexts: reactive dialogs / prompts (HATEOAS / HAL protocols).

Model Services: Browse DOM layers. Monads parsed DOM interactions services (functor contexts) available as operations over rendered models (HATEOAS).

Interactions: Services. Browse DOM. Apply selectors / browse available transforms (Monads / HATEOAS). Monads applications render / update DOM / HATEOAS browsing response.

Outputs. Connectors / Services (active Resource topics). Feedback (Events Inputs).

**Deployment:**

Apache MetaModel. JBoss Teiid. Connectors (I/O). APIs: Model Services (reify data, schema, behavior alignment in Connectors data structures). OpenRefine Knowledge (data, schema, behavior) alignment extensions (Model Services APIs). Knowledge transactions (inferred "wizards") contextual wiki like augmentation: Apache Stanbol (guided assistance). Context Wiki: JCR XPath / Apache Clerezza.

**APIs:**

Contexts (DCI / HAL / HATEOAS):

Context Guided Data augmented (contextual hypermedia) Interactions. Wizards APIs. XForms: rendering (REST HATEOAS).

**Service Resources:**

Service Resources. ContextResource scoped prediction generalizations (encodings):

Meta Meta Model: (Relation, Statement : Entity, Kind : Relationship, Resource : Flow);

(PredictionService : Relation, Context: Entity<PredictionService>, Features : Relationship<Entity>, Output : Flow<Relationship>);

Naming Service:

Synsets, generalization/specialization term rels:

(NamingService, Context, TermRel, Term);

Registry Service:

Hierarchical key / value (property graph):

(RegistryService, Context, Key, Value);

Index Service:

Apache Lucene, Vector Space Model Triple / Quad polygon encoding:

(IndexService, Context, Term, Result);

IOService (Connectors):

(IOServiceConnector : Relation, ContextResource : Entity, Attribute : Relationship, Value : Flow);

Relation: Aligned Context.

Entity: Aligned Resource.

Relationship: Aligned Schema.

Flow: Aligned Behavior (schema resource data flows).

ToDo: Resource selector API.

ToDo: Transforms data flow. Application activation (available operations / state signatures / events). Browse (HATEOAS), reactive / event driven protocol / dialog. Applicable transforms (example: IO Connector updates).

Domain Service (ActivationService):

ActivationService:

Services which performs addresses / body request activations (ResolutionService).

(Domain, Flow : Flow, Role : Relationship, Item : Entity);

Flow: Use Case. Produced / available for Domains.

Role: Prompts. Produces / available in state of Flow. Item: product, prompts consumer / producer kind.

Item: Product / Good / Need (Goal). Produces / populated by Roles (Prompts).

(DomainService : Relation, Resource : Domain, Context : Role / Relationship, Item : Entity);

Retrieves available workflows Items for Roles in domain.

**Upper Ontology. Grammars. Primitives:**

Primitives:

Primitive Resources for primitive Kinds for primitive Statements for primitive Relations.

Primitive relations example: opposite, inverseOf, causeEffect, etc.

Grammars:

Resources productions for Kinds productions for Statements productions for Relations productions.

Upper Ontology: Primitives Grammar.

**Ontology Matching:**

Data, Schema, Behavior matching / alignment.

Data: keys / values.

Schema / Information: relation tuples rows.

Behavior / Knowledge: relation tuples rows data / information flows (dimensional).

FCA Augmented Resources.

See Protocol Addressing / Matching.

**Protocol. Message IO. Dataflow:**

DOM Hierarchy:

Resource<OntResource[]>;

Kind<Resource[]>;

Statement<Kind[]>;

Relation<Statement[]>;

Entity<Relation[]>;

Relationship<Entity[]>;

Flow<Relationship<Entity[]>;

Domain<Flow[]>;

Addressing / Matching:

Message Event dispatch. Message is an address plus browse state representation.

Matching. Distributed Addressing Encoding:

Encoding: Encode metaclass, class, instances, occurrences (contexts) in addresses.

Identifiers assigned according context objects and metadata "paths" following a pattern or "shape" in a way analogies can be inferred (Monad Zippers) of its (nested / linked ) metaclass, class, instances, occurrences "trees" (cons cells).

Matching: Same objects resolve to equivalent addresses in different models when addresses follows / match Zippers shape.

Populate ResolutionService Relations. Zippers paths / shapes specialized / generalized matches (concepts hierarchies). Others services metadata.

ActivationService:

Services which performs addresses / body request activations (ResolutionService).

(DomainService : Relation, Resource : Domain, Context : Role / Relationship, Item : Entity);

ResolutionService (Matching):

(ResolutionService : Relation, Resource : Entity, Model : Relationship, Resource : Flow);

Resulting Flow: next state (activation) over browse representation request.

ActivationService:

Referrer.

Address.

Body.

Next State.

(ActivationContext : Relation / Referrer, Statement : Body, Kind : Address, Resource : Next State);

ActivationService Resource Object (Resource) is next / matching Resource for request context (Referrer, Address, Body).

Activation Patterns (DOM Layers hierarchy matching / Layers reified Resources matching):

(Resource, Role,

(Kind, Resource, Role,

(Statement, Kind, Resource, Role);

(Relation, Statement, Kind, Resource);

(Entity, Relation, Statement, Kind);

(Relationship, Entity, Relation, Statement);

(Flow, Relationship, Entity, Relation);

(Domain, Flow, Relationship, Entity);

Semiotic / Dimensional Roles Facets: Idem for equivalent Relation hierarchy layer.

Monads: Reify available Transforms as activable Resources (Function addresses). REST / HATEOAS HAL.

Prompts / Dialogs: Function arguments (values / options) shown as link addresses in Transforms navigation Flows. Activation browse of Resources in Transform context.

Chained Activations for complete contexts resolution / flows. Complete layers productions rendering / navigation from higher to lower layers.

Encoding / Parsing:

Representation: Message Event production / consumption.

Abstract Content Type: Render Model DOM Hierarchy.

Browse: request address content representation (extracted from current state) embedding current state representation as request context body. Model matches address and returns augmentation using request body as argument / context.

Sets / Individuals Mappings:

IDs: metaclass, class, instance, context, role, occurrence, previous, next ID roles relations for Model Set Contexts.

Augmentations / Transforms: Model / Domains functional mappings. Order. Dimensions. Axes. Flows. Hierarchies. Inference / Population.

Levels: Augmentations. Mappings.

Levels: Resource, Kind, Statement.

Levels: Reify Statement as Kind, Kind as Resource, Resource as Statement.

Levels: Reify Resource as Kind, Kind as Statement, Statement as Resource.

Sets / Individuals Mappings:

Levels (layer statements) shifts (quads matrix). CSPO roles:

(Dimension, Resource, Kind, Statement);

**Dataflow. Layers:**

Sets Aggregation:

Sets: Context / Role, Resource, Kind, Statement. From ActivationService / IOService raw RDF / events response statements (feedback). FCA / Transforms synchronization (events / signatures).

Layers: Context / Role, Resource, Kind, Statement, Relation, Entity, Relationship, Flow, Domain. Quad store mappings (functions). Sets / FCA / Transforms synchronization.

FCA Context Aggregation:

Populate concepts objects / attributes from layers mappings. Sets / Transforms synchronization (events / signatures). Sets / DOM / Transforms synchronization.

Resources belonging to multiple sets / concepts: degree of pertenence. Concepts hierarchy specializations (Kinds).

Objects: Layer Context Resources.

Attributes: Layer Contexts scaled to their CSPO (Context, Occurrence, Attribute, Value) Resources.

DOM Functional Aggregation:

Instantiate DOM Layers Monads from FCA Context (Wrapper type: Layer Context Role, values: FCA Objects) / Transforms (FCA Concepts. Matching concepts signatures reified into Resource functions transforms mappings. Concepts lattice flows. Aggregation: lattice concepts (transforms) / objects (functor values) / attributes.

Transforms (Resource functions transforms mappings) populated with possible source / dest values from context concepts objects / attributes. Concepts flows available when Resource matches source attributes. Transform Resources available for each DOM layer.

Example: 'anEnterprise' Entity Monad. Available flow / transform: concept Resource: 'Corporation'. Reified lattice concepts as Resource functions transforms mappings (source / dest mappings).

Transform application merges / translates Entity Relations with applicable mappings from Resource functions transforms mappings (of which concept Resource has source / destination transform mappings).

Example: Entity 'aCorporation', concept Resource 'CEO', Entity with merged attributes from Resource functions transforms mappings.

Protocol. HATEOAS. Available transforms flows rendered as browseable Resources. CRUD: Browsing values for a Resource mapping transform has REST semantics for activating concepts with new Resources.

Selectors: matching / activation: match Monads functors by their attributes (signatures), apply transform (mapping function resource transform request address) over referrer body (yields next state functor). Events: Monad functors listen matching browsing events and publishes transform results (ActivationService IO streams). Update DOM / Layers. FCA / Sets synchronization (events / signatures).

Context Resource functions transforms mappings (FCA Concepts Augmentation Layers):

(Object : Resource, Concept, Attribute : Resource, Value : CSPO Role);

Object Resource Role type: Monad wrapper layer type. Object Resource: Monad values (Resource subjects in concept context).

(Flow, Object : Resource, Concept, Attribute : Resource);

Flow queries / prompts for Attributes. Update context / augmentations.

TBD.

**To Do:**

Context Sets. Layers Aggregation (MapReduce).

Backends (Layers Repository, Services):

Property Graphs. Graph algorithms (encodings). Functional APIs: Event Driven Facade (Reactive Services).

Stores: Jena (Fuseki Services), RDF4J (N3, RIO, SAIL), Neo4j.

Fuseki: SPARQL Services.

Models: RDFS / OWL (Jena). Reasoning.

Schema: SHACL, ShEx. Matching.

Protocols / Formats: XML RDF / OWL, JSON-LD, Turtle, N3.

Endpoints: SPARQL, JSON-LD / HAL, GraphQL. Queries / Templates (HATEOAS HAL Forms / Flows).

FCA: From federated Peers / Endpoints layers navigation representations.

DOM: From FCA Context Lattice. Client high level DOM representation APIs (reactive / streams): navigation, transforms. Client I/O: DOM representations navigation (browse / transforms).

Endpoint DOM representation navigation: Standing in a browsed address body (referrer) render address context: nested context aggregated SPOs (DOM Monad values), reified HATEOAS Transforms as context parent contexts navigations (browsing shows Transform attributes prompts as browseable HATEOAS).

JSON-LD REST HATEOAS / HAL / GraphQL DOM wrapper Endpoints. SOAP Endpoints. APIs / Interfaces (objects / schema / behavior) inferred from DOM models. Discovery (workflow contexts state flows) through DOM metadata.

FCA / VSM (Vector Space Model) Encoding:

Attributes: Resource URIs. Polygon side lengths (class).

CSPO Roles (scaling): polygon sides (metaclass).

CSPO scaling: ordered side position.

Polygon sides dot-notation ordered sides lengths: Resource Layer Statement IDs (instance).

Sides dot-notation sum: side in context (occurrence).

Normalization: Resource URI attributes embeddings / primes quad polygon sides lenghts.

Nested Resource encoded attribute values (layers hierarchy): sides lengths concatenation (ordered dot notation) sum (occurrence).

Graph navigation (layers / transforms: concepts / objects containing / contained in concepts / objects attributes IDs / lengths).

Distributed Contexts (label / tag metadata statements) and Versioning: Blockchain / Git / Apache Kafka persistence. Event sourcing / DIDs. Distributed back ends / data sources.

Dataflow layers:

Sources / Persistence: Reference Model encoding. Event sourcing. Ontology Matching. Resolvable DIDs.

Sets (Layers Roles) Population / Aggregation (Layers: MapReduce).

Triple Store / Model Resource Layers.

FCA Contexts. Encoding. Flows. Order (types: dataflow signatures domain / range, instances: dimensional attributes).

Functional DOM: Meta Model Resource Layers Functional DOM (FCA / Monads / Transforms)..

Forms / Flows: DCI HATEOAS DOM Functional Protocol Client APIs.

Tabular / Object Type Object pattern DOM / JAF APIs. Graph DTOs (REST).

Events: Reference Model encoded streams (Sources to / from APIs / IO).

Meta Model (Layers / DOM):

CellValue

ColumnField

ID : occurrence (PK)

Context : instance (table)

Role : metaclass (CSPO)

Resource : class. Monad Value (instance)

Kind : selector / transform (Functor mapping). Monad Value Type (metaclass / role)

Statement (context)

Relation : Kind Grammar (Productions). Monad Instance (occurrence)

Entity : Kind Grammar (Rules). Monad Type (class)

Relationship

Flow

Domain

Monad and Grammar roles apply for reified layers Resources.

Messages:

Rules: Entity (grammar: infer available messages).

Productions: Relation (infer / parse Messages).

Matching: Kind (selector : Entity Relations).

Transforms: Kind (transform: Relation Productions of Entity Kinds).

Dataflow: Result Transform matching rules signatures.

Layer::flatMap(attr : Kind) : Layer;

Message: Graph layer statements(s) populated with Relation (Productions) nested into Entity Rules to be applied / applicable to the Message Relations (Productions). Relations with concrete Resources or Kind matching model layers instances. Existing or new Production: Resource or empty Kind results. Update / Delete: override previous version.

Build Message graph via navigation of the model (Forms / Flows HATEOAS APIs, Kinds domain / range). Transform mapping: Kind prompts: apply Rule Kind to Production Resource: Productions.

Message: Relation statements. Productions.

Message: Rules. Relation statements of Rule Kinds on to operate over Relations of Kind.

Create: Relation not matching existing one. Rules Kinds Productions populated prompting Message Relations / Model with Rules Kinds.

Update: Update Model Relation Resource matching Message Relation of Rule Kind updating Relation Resource of Rule Kind. Update references. Versioning.

Delete: Relation Kind in Rules but not in Productions. Update references. Versioning.

Retrieve: Relation Resource of Rule Kind matches model Kind Resource. Prompts.

TBD: Monads, Layers, FCA, Sets, Persistence synchronization. Events.

TBD: Encoding, Model, Protocol (Messages DCI Forms / Flows):

Data Model: DOM / JAF. Object / Tabular Type Object pattern. OGM. TMDM.

Reference Model: Cons lists / FCA nested context pairs / contexts (Link Grammar constraints). (Type : Value). TMRM.

To do:

Render synchronization / consistency across dataflow layers via Meta Model DOM / Messages. Reify layers into Meta Model. Resource layer implementations (context URI) invoked / invoking Resources with Resource Message populated (encoding) with event upper layers values.

Upper / Onto Matching: reify Resource upper layers as Resource and aggregate into lower layers. Reified Entity, Relationship, Flow, Domain as upper layers and aggregated downwards (Rules / Productions). Productions dataflow (domain / range).

Ontology Matching:

Matching: Resource occurs as context / occurrence / atribute / value or class / occurrence / context / metaclass / instance in equivalent occurrence contexts (kinds / order / shapes / type hierarchies).

Meta Model encodes mappings for equivalence / relations hierarchies for entities instance occurrences in roles in contexts for concepts recursively till upper onto / primitive terms / relations.

Workflows (Domain Goals) general purpose ontology matching integration framework.

Workflows (Domain Goals) general purpose ontology matching integration framework.

Components:

Reference Model (Component Message Adapters). Component Monads of Component Nodes Functional events (bus) dataflow (selector signatures).

I/O. Persistence. Events. DIDs Components Nodes

Sets Component Node

Layers Augmentation: Aggregation (layers), Alignment (ontology), Activation (dataflow) Component Node

Layers Quads Component Node

Triple Store Component Node

FCA (Monads AST Builder. Updates Quads Productions) Component Node

Layers Monads / Parser Monads (Messages : Rules / Productions). Functional events dataflow (selector signatures : Activation) Component Node

Forms / Flows (Grammar / Protocol Builder. Prompts) Component Node

Augmented Resources Contexts / Interactions Services Component Node

OGM / Client Drivers Services Component Node

Services / Mappings:

Upper Ontology. Grammars. Primitives. Ontology Matching

DOM Hierarchy:

Resource<OntResource[]>;

Kind<Resource[]>;

Statement<Kind[]>;

Relation<Statement[]>;

Entity<Relation[]>;

Relationship<Entity[]>;

Flow<Relationship<Entity[]>;

Domain<Flow[]>;

Meta Model (Layers / DOM):

CellValue

ColumnField

ID : occurrence (PK)

Context : instance (table)

Role : metaclass (CSPO)

Resource : class. Monad Value (instance)

Kind : selector / transform (Functor mapping). Monad Value Type (metaclass / role)

Statement (context)

Relation : Kind Grammar (Productions). Monad Instance (occurrence)

Entity : Kind Grammar (Rules). Monad Type (class)

Relationship

Flow

Domain

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.

**Monads.docx**

Type constructor (wrapper / type).

Unit wrapper / value instance.

Bind / map / flatMap. Instance argument, transform (static / functor). Instance method (applications).

Functor: A category consists of a collection of nodes (objects) and morphisms (functions). An object could be numbers, strings, urls, customers, or any other way you wish to organize like-things. (X, Y, and Z in the graphic are the objects.).

A map is a function to convert something from one object to another. (f, g, and fog are the maps). Google tip: A map between objects is called a Morphism.

So an object could be simple like a Number or a String. An object could also be more abstract like a Username, A User API URL, User API HTTP Request, User API Response, User API Response JSON. Then we can create maps or morphisms between each object to get the data we want.

Examples of morphisms: Username -> User API UrlUser API Url -> User API HTTP RequestUser API HTTP Request -> User API ResponseUser API Response -> User API Response JSON

Google tip: Function Composition is a way to combining multiple map or morphisms to create new maps. Using Function Composition we could create a map from Username directly to User API Response JSON.

Now that we understand what it means to be Mappable, we can finally understand what a Functor is.

A Functor is something that is Mappable or something that can be mapped between objects in a Category.

An Array is Mappable, so it is a Functor. In this example I am taking an Array of Numbers and morphing it into an Array of Strings.

Note: One of the properties of a Functor is that they always stay that same type of Functor. You can morph an Array containing Strings to Numbers or any other object, but the map will ensure that it will always be an Array. You cannot map an Array of Number to just a Number.

We can extend this Mappable usefulness to other objects too! Let's take this simple example of a Thing.

If we wanted to make Thing mappable in the same way that Array is mappable, all we have to do is give it a map(morphism) function. And that is a Functor! It really is just that simple. Google tip: The "Thing" Functor we created is known as Identity.

Monad:

Sometimes functions return a value already wrapped. This could be inconvenient to use with a Functor because it will re-wrap the Functor in another Functor.

This is where flatMap comes in handy. It's similar to map, except the morphism is also expected to perform the work of wrapping the value.

Summary:

A Functor is something that is Mappable or something that can be mapped between objects in a Category.

A Monad is similar to a Functor, but is Flat Mappable between Categories.

flatMap is similar to map, but yields control of the wrapping of the return type to the mapping function.

Monads:

Monads are a way to compose type lifting functions: g: a => M(b), f: b => M(c). To accomplish this, monads must flatten M(b) to b before applying f(). In other words, functors are things you can map over. Monads are things you can flatMap over:

A monad is a way of composing functions that require context in addition to the return value, such as computation, branching, or I/O. Monads type lift, flatten and map so that the types line up for lifting functions a => M(b), making them composable. It's a mapping from some type a to some type b along with some computational context, hidden in the implementation details of lift, flatten, and map:

Functions map: a => b which lets you compose functions of type a => b

Functors map with context: Functor(a) => Functor(b), which lets you compose functions F(a) => F(b)

Monads flatten and map with context: Monad(Monad(a)) => Monad(b), which lets you compose lifting functions a => F(b)

Map means, “apply a function to an a and return a b". Given some input, return some output.

Context is the computational detail of the monad’s composition (including lift, flatten, and map). The Functor/Monad API and its workings supply the context which allows you to compose the monad with the rest of the application.

The point of functors and monads is to abstract that context away so we don’t have to worry about it while we’re composing things. Mapping inside the context means that you apply a function from a => b to the value inside the context, and return a new value b wrapped inside the same kind of context.

Observables on the left? Observables on the right:

Observable(a) => Observable(b).

Arrays on the left side? Arrays on the right side:

Array(a) => Array(b).

Type lift means to lift a type into a context, blessing the value with an API that you can use to compute from that value, trigger contextual computations, etc… a => F(a) (Monads are a kind of functor).

Flatten means unwrap the value from the context. F(a) => a.

Dataflow, Reactive: Function composition creates function pipelines that your data flows through. You put some input in the first stage of the pipeline, and some data pops out of the last stage of the pipeline, transformed. But for that to work, each stage of the pipeline must be expecting the data type that the previous stage returns.

A monad is based on a simple symmetry A way to wrap a value into a context, and a way to unwrap the value from the context:

Lift/Unit: A type lift from some type into the monad context: a => M(a)

Flatten/Join: Unwrapping the type from the context: M(a) => a

And since monads are also functors, they can also map:

Map: Map with context preserved: M(a) -> M(b)

Combine flatten with map, and you get chain — function composition for lifting functions, aka Kleisli composition:

FlatMap/Chain Flatten + map: M(M(a)) => M(b)

Monads must satisfy three laws (axioms), collectively known as the monad laws:

Left identity: unit(x).chain(f) ==== f(x)

Right identity:[m.chain](about:blank)(unit) ==== m

Associativity:[m.chain](about:blank)(f).chain(g) ==== [m.chain](http://m.chain)(x => f(x).chain(g)

Functor: mapping between categories A / B using a function. Map, alcance, rango, dominio, imagen (infer, aggregate). Connections: same number of items (inyective, biyective). Function: de wrapper en wrapper (morphism new image). Message / Augmentation Metamodel reifications.

Actor / Role. Dynamic Object Model (OGM / Kinds). Golden Braid: Metaclass, class, instance, occurrence relation relative to layer context levels.

Reference Model (DOM / Actor / Role OGM):

(Relation, Statement, Kind, Resource);

Type : Value;

(Value, Type);

(Type, Type);

(Value, Value);

Aggregation: cons cells tree traversal. Layers contexts representation. Dataflow augments updates / append corresponding tree cells: (first: (rest: nil).

(C (S (P (O))));

Aggregate / Augment inputs / transforms: Parsed model streams. Reactive data structures.

Encoding: addressing, semantic graph networks. URN overlay semantic addresses encoded mappings.

Dataflow:

Input statements: Augment. Aggregate, build layers contexts representation quads. Parse:

Monadic parser combinator: aggregated metamodel nested layers contexts corresponding wrapper / wrapper hierarchy types DOM / AST (from contexts quads aggregation nesting hierarchy levels).

Augment: signatures (case meta / classes / instance / occurrences) dataflow. Apply input statements: new inputs / kind new attributes (person, employee -> position, salary). Parse.

Protocol: Forms / Flows augmented / parsed representation / metamodel I/O. HAL / HATEOAS endpoint encoding for navigation, transforms and inputs augmentation.

Protocol: Forms / Flows Dialog / Prompts resolution. Context roles, wrapper kinds navigation / transforms declaratively stated in encoded representations.

Monad: context type. Metaclass.

Monad: wrapped type. Class.

Monad: wrapped value. Instance.

Monad: wrappers hierarchy context type instance. Occurrence.

Unit. Type constructor (hierarchy context / member types / values factories). flatMap / map / flatMapN.

Monad: context type case classes hierarchy? Factory methods, hierarchy types / members values / signatures (wrapped context / values) cases (predicates):

Optional: None | T [case context | case context / case value signature]

Result: Error | OK [case context | case context / case value signature]

Writer: Value, List<S> trace

Monads: Quads Contexts wrapper of Occurrence / Subject aggregations. Root type: Resource. Kind (model / semiotic / domain meta resources) functors filter / traversal (streams: flatMap Resource set specifications).

Wrapped resources holds references to its wrapping occurrence contexts (Resource root type interface, contexts hierarchy / reification levels: upper layer instances reify layer occurrences in contexts).

Wrapping unit / bind: layers traversal until container wrapped category type is met. Example: Behavior(s) of an Entity. Augmentations (types, roles, attributes reified model / domains aggregations). Example: Model, SK, P; Domain, someSubjKind, somePredicate.

Form / Flow: state navigation / browse. Dialogs / Prompts (contexts). Messages I/O: Augmentation signatures (kinds) streams.

Messages: Augmentation according signatures. Broadcast. Key / Value, Event sourcing routes (levels). Form / Flow Dialogs / Prompts Messages  for signatures (kinds) of unknown Resources resolution (layers wrapping traversal, CSPOs layers resolution for CSPO signatures / attributes / values).

Encoding: IDs. Embed metaclass, class, instance, occurrence metadata (context, role, attributes, values). Functional APIs. Wrappers / Transforms (augment: aggregate / classify, roles, properties "graph" rels). Polygon Vector Space Model. ANN embeddings / autoencoders.

Forms / Flows Dialogs / Contexts. Protocol. Resources, addressing, representations, navigation / traversal: properties "graph" rels (Wrappers / Transforms). Functional APIs.

Sets encoding: properties in axes (kinds). SortedSet (hierarchies). Metaclass, class, instance, occurrence properties in axes for CSPO IDs. Augmentations: property graph rels navigation / traversal. Dialog Forms / Flows "state" contexts.

Contexts Wrappers kinds Transforms / Traversals functors: Augmentations declaratively stated in upper Context layers (kind classification, kind roles, kind attributes / values).

Dialog Forms / Flows "state" Contexts browsing (upper Context SPO kinds: current context streams).

Augmentation navigation of Transforms / Traversals as a Context (streams / filters). Levels / reification.

Order. Iteration. Predicates (resource meta / domain / kinds). Streams filter, conditionals, jumps. Aggregation. Functional mapping / reduce, etc. Dataflow signatures (case meta / classes / instances /occurrences)

Resource API (layer roles):

Statement: Resource (CSPO: Statement) Property graph URL wrapper. URL occurrences aggregate. Functional occurrences properties (roles / streams) for Statement wrapped URL:

Contexts: Metaclass URL occurrences.

Subjects: Class URL occurrences.

Predicates: Instance URL occurrences.

Objects: Occurrence URL occurrences.

Statement wrapped URL occurrences functional roles Kinds: Map<Role, Kind> (reified in Metamodel).

CSPO Kinds (streams functors) declaratively stated in aggregated layers CSPO occurrences (kind classification, kind roles, kind attributes / values).

Augmentation navigation of kinds Transforms / Traversals as streams / roles (filters). Levels / reification: kinds from Statement / roles layers (reification / levels axes).

Query / stream context selectors: ID (URL), layer context type, layer context role, layers kinds. Transforms (functor kinds: augment / browse query context according kind specification with corresponding statements).

Levels: Grammars (kinds functors dataflow signatures). Productions: Augmentations (parsed / produced in navigation contexts). Dataflow order (sets / hierarchies).

Aggregation: inputs in contexts layers. Case matching: metaclass, class, instance, occurrence, kind grammars.

Parsing: Monadic combinator parsers: quads contexts layers (recursion). Metaclass, class, instance, occurrence, kind aggregations parsers.

Augmentation. Functional API: Monadic DOM / AST Parse Tree (cons cells) context layers hierarchy wrappers, Resource metaclass, class, instance, occurrence, kind hierarchy wrappers (i.e: contexts instances / parsed kinds).

Dataflow: Aggregation, Parsing, Augmentation. Streams: reactive / event driven. Model reified Message Functors / Transforms.

Zippers: Aggregation / Addressing: Locations / Contexts. Parsing. Monads. Augmentation (navigation / transforms) Reactive Streams (location observers / observables: paths / kind paths dataflow signatures).

Inputs / CRUD navigation / transforms Augmentation Dataflow: parsing / zippers over core Model reified context layers meta Resources AST / DOM parsers (zippers paths in meta Resource aggregates / parses inputs in contexts). Resource metaclass, class, instance, occurrence paths / cases matching zippers.

Blockchain: P2P (JXTA / Git) DL Distributed Ledger inter Node Backend. DIDs (traceable semantic state: Distributed IDs encoded / embedding result of transforms, labeled / property graph statements / contexts: saga / zippers). Smart contracts (signatures: Dataflow). Monads (immutable state, transactions: functor morphisms). Zippers (chain contexts: mutable chain branches, dimensional contexts / labeled property graphs). Reactive Augmentation (I/O) APIs: Resource metaclass, class, instance, occurrence paths / cases case matching (inputs quads, parsed DOM, outputs quads). Quads Forms / Flows Protocols.

Node Protocol: Forms / Flows DAV HAL / HATEOAS Client Application Sessions (navigation contexts). JCR, Hierarchical structures (XML, XPath, XSL, XLink, XQuery, XPointer) representations of augmented reactive DOMs. Representation Levels (onto meta resources): metamodel / session / domains. Behavior encoded in (augmented) representations functional contexts traversals. JXTA / Git Backend inter Node P2P Blockchain Node quads DL IO sync. Connector Nodes: reactive dataflow (signatures: smart contracts).

Augmentation

Aggregation, Alignment, Activation:

Class (relation) and instance (relationship) being the things that could be asserted for each (domain / range for classes, pairs of "roles" for instances and attributes for both: as property graph) the difference between relation and relationship.

A naive approach of render this in pseudo RDF / RDFS:

Marriage : Relation;

Husband domain Marriage;

Husband range Male;

Wife domain Marriage;

Wife range Female;

Marriage properties (date, etc.);

aMarriage : Marriage;

aMarriage husband Pete;

aMarriage wife Mary;

Marriage attributes (domain / range). Reified Relation instances entails statements (expands links, attributes in property graphs) for Relationship roles / players attributes:

Peter marriedWith / husbandOf Mary; domain: spouse / husband; range: spouse / wife;

Mary marriedWith / wifeOf Peter; domain: spouse / wife; range: spouse / husband;

marriedWith / husbandOf / wifeOf statements in a CSPO context: aMarriage;

There should be an inference method materializing inferences of role instances attributes according the Relation class Relationship instance roles they play.

Or, if RDF Quads are not available, entailed properties schema / instances: marriedWith:\_0 / husbandOf:\_0 / wifeOf:\_0 instances of corresponding relation class attributes. Entailing relationship (aMarriage) instances context attribute.

marriedWith:\_0 rdfs:type / rdfs:subPropertyOf marriedWith (expansion property kind).

The case is that a "terminals" relation relationship resource statements "expansion" materialized view renders the Relationship "extension". Way back entailing / inferring relation / relationship class / attributes roles should be possible.

Source (higher order) relations may relate relations / relationships with other relations / relationships thus allowing a richer set of concepts into an ontology / dialect. Example: Peter / Mary Husbandhood related to their Marriage. RDFS domain / range properties provides the inference means here to parse such a relationship entailing relation context / roles.

Rules for expansion: if a Relation is a class for Relationship(s) which has Role(s) for Resource(s) in SPO statements the statements expansion is the "materialized" view of the Relation instance in SPO statements.

Having a tuple:

(Template, Relation, Relationship, Role, SPORole, Resource);

Aggregated Template SPORole Resource should enable the use of some query mechanism (SPARQL? Zippers?) for building output triples. Aggregation intension / extension bidi transform.

Reify from lower layers to expanded statements materialized views and aggregate forward (I/O, Augmentation).

Transform: apply Kind Relation. Relation defined by extension (tuples) and intension (property / attributes relations).

Layers hierarchy:

(Context, Object, Concept / Sign, Value);

(Resource, Resource, Resource, Resource);

(Role, Resource, Resource / Attribute, Resource / Value);

(Statement, Role, Resource, Resource / Attribute);

(Entity, Statement, Role, Resource);

(Template, Entity, Statement, Role);

(Mapping, Template, Entity, Statement);

(Flow / Augmentation, Mapping, Template, Entity);

(Behavior / Message, Flow, Mapping, Template);

(Measure, Behavior, Flow, Mapping);

(Unit, Measure, Behavior, Flow);

(Dimension / Axis, Unit, Measure, Behavior);

(Value, Dimension, Unit, Measure);

(Concept, Value, Dimension, Unit);

(Object, Concept, Value, Dimension);

(Context, Object, Concept, Value);

Reify from lower layers to expanded statements Resource materialized views and aggregate forward into relations / relationships: contexts (I/O, Augmentation).

(husband, role, resource, ?);

(aMarriage, husband, role, resource);

(Marriage, aMarriage, husband, role);

Relation / role type promotion. Contexts. Augmentations (of promoted players role kinds transforms): relationship and expanded members / attributes / links / relations.

Relation<Relationship<C, S, P, O>> (CSPO : Relation) Monads root hierarchy.

Dataflow:

Monads / Zippers (cons / graphs). Aggregation, recursion. Expressions. Signatures.

Aggregation: nesting. Relationship C Relation holding same C context role corresponding / prefix of aggregated SPOs, same CSs for aggregated POs, etc.

Relationship: Kinds / Roles. Aggregations: traversal / expressions (bound functions renders CK, SK, PK, OK).

Parent layer: current layer extension / expansion.

Current layer: C intension, O extension.

Next layer: current layer intension.

Dataflow: perform augmentations on layers instantiations. Observers, observables, signatures.

Inferir relación dominio / rango, alcance / campo. Describir relacion n-aria como predicados.

TBD.

Model Hierarchy:

Resource : Relation. Relationship CSPO: metaclass, class, instance, occurrence Resource Relation context roles.

Relationship CSPO: Context, Kind, Statement, Role, Resource reified Relation(s): Resource Relationship(s) instances (aggregation). Metaclass, class, instance, occurrence.

Dimensional: Events. Causal, roles (marriage). State, predicate properties from / to (single / married: marriage, married / single: divorce). Actor / Class / Role: metaclass, class, instance, occurrence. Marital status example.

Relation by expressions / predicates: brotherhood(a.parent = b.parent). Predicates linking (actor / class / role) to a dimensional event.

Reify Attributes / Values as Relations (Relationship Kinds instances).

ISO:

About Relationship: many Relationship instances are the result of an Activity, e.g. Marrying – Marriage, Assembling – Assembly, Containing – Containment, Connecting – Connection, Employing – Employment, etc.

We model that by typing a Relationship with a (meta) ClassOfRelationshipWithSignature that is defined as a ClassOfRelationshipWithSignature is a ClassOfRelationship that may have a RoleAndDomain specified for each end. (where RoleAndDomain simply stands for ‘a Class in a Role’)

The instance of ClassOfActivity CONNECTING-A-TRAIN

:CONNECTING-A-TRAIN rdf:type dm:ClassOfActivity .

:CONNECTING-A-TRAIN :hasPartiipant1 RoleAndDomain1 .

:CONNECTING-A-TRAIN :hasPartiipant2 RoleAndDomain2 .

:RoleAndDomain1 rdfs:subClassOf rdl:LOCOMOTIVE .

:RoleAndDomain1 rdfs:subClassOf rdl:PULLER .

:RoleAndDomain2 rdfs:subClassOf rdl:TRAIN WAGON .

:RoleAndDomain2 rdfs:subClassOf rdl:PULLED .

The instance of ClassOfRelationshipWithSignature

:CONNECTION-OF-A-TRAIN rdf:type dm:ClassOfRelationshipWithSignature .

CONNECTION-OF-A-TRAIN :hasClassOfEnd1 RoleAndDomain1 .

:CONNECTION-OF-A-TRAIN :hasClassOfEnd2 RoleAndDomain2 .

The typed Relationship

:myRelationship rdf:type tpl:CONNECTION-OF-A-TRAIN ;

:myRelationship :hasPuller myLocomotive ;

:myRelationship :hasPulled myTrainWagon .

An instance of Relationship, typed with this metaclass CONNECTION-OF-TRAIN, can be linked to an instance of Activity, typed with ClassOfActivity CONNECTING-A-TRAIN, with an instance of above CauseOfEvent.

When I connect a train I cause the Event ‘train is connected’, which leads to a state that the locomotive and the trainwagon instances are connected, a fact that is recorded with an instance of ConnectionOfTrain relationship.

Dataflow:

Data / Information / Knowledge. Levels. Formalization. Reference Model. Transforms:

Aggregate / Deaggregate CSPO (expand / collapse intension / extension) dataflow. Layers in / out traversal:

Class (prev): (Class, Subject, Property, Object);

Instance (this): (Value, Context, Concept, Object);

Occurrence (next): (Value, Dimension, Unit, Measure);

Levels: aggregate dimensional context properties. Relation levels: data / info / knowledge expansion. From DCI / actor role / ontology "use cases" (rendered "real world" application "behavior") to fine grained Resources CSPO Statements.

Data: product price / marital status.

Information: price variation / state change.

Knowledge: increase, decrease / marriage, divorce.

TBD.

Relationships: verbs (infinitive), relation: verb (conjugation, CSPO context roles). Verb: action, passion, state (of roles).

(C[context;], ,S[action; role: schema; player: data], P[state; verb: domain/range; mappings: properties], O[passion; role: schema; player: data]);

(data:schema, behavior:state, schema:data);

Graph Normalization (kinds). Levels. Direction (labels): fatherOf, sonOf (property types: inverse of, reflexive, simetric, transitive, etc.). Reification (kinds type properties / values implies relationship instance (salary: Employment, same sonOf value: brotherhood), relationship instance implies kinds attributes / values).

Dimensional: from possible knowledge to information to data. From actual data to possible information to knowledge. Order: from types promotion (domain / range).

Knowledge: Functor: Ownership (Person : owner, Dog : owned). Relationship

Information: Morphism: Owns (Peter, Fido). Relation (anOwnership). Relationship roles promotion. State change / events (reified relation state).

Data: :ownsDog, :owner. Attributes / Values. State: to / from relationship relation attributes / values).

(Knowledge, Information, Attribute, Value);

Objective: achieve normalized form from which aggregate data into layered (data / information / knowledge) layer occurrences relationship (layer roles relations) and enable further knowledge to be de aggregated into its information and facts. Layers specialization in each part of the model.

Dataflow: Reactive DOM. Events. Augmentations. Streaming I/O (signatures, domain / range roles ordered pipelines selectors). Model events bus.

Resources (top layer input): populate / augment / align lower context layers (metamodel / upper onto / kinds placeholders). Render Statements, Roles, Entities, Interactions, Contexts / Mappings, Flows / Behaviors, Dimensional and Semiotic aggregations to be populated / augmented via further input facts (Grammar).

Dimensional / Context (bottom layer input): base upper ontology resources / browse / roles prompts till base Resource layer (facts).

Metamodel subscriptions: Reified (static) layers contexts observes / observable of upper layers types, observer / observable of lower layers types Model events. Matching instatiate contexts.

Domain subscriptions: Context layers (instances) observes / observable of upper layers instances, observes / observable of lower layers instances Model events. Matching augments contexts.

Example: Data (relation / facts: Entity), Information (events: Template / Interaction), Knowledge (relationship: Mapping / Context) layers abstractions.

Browse: Form / Flow. Selectors / zippers. CRUD (HATEOAS / HAL APIs).

Browse Streams Dataflow: Data / Information / Knowlege streams. Upper: SP/CS selector (intension contexts statements) expands lower layer information / facts. Lower: CS/SP selector (extension object statements) aggregates upper layer information / knowledge.

Dataflow: Metamodel (reified / static) contexts subscriptions. Contexts monads hierarchy (Relationship).

Dataflow: Model (domains / instances) contexts subscriptions. Contexts types hierarchy (Relation).

Inputs: Facts (data). Aggregate. Dataflow events.

Inputs: Relationship (knowledge). Populate facts (roles prompts). Dataflow events.

Browse: expand facts (till input data).

Browse: aggregate relationships (till aggregated knowledge).

Browse Dataflow: Selectors. Available roles input kinds (apply lower layer relationship). HATEOAS. Transforms / mappings / functions / contexts as functions.

Input Dataflow: Apply Kinds. Role promotion. Knowledge input aggregation. Data facts prompts. CSPO Dataflow: monadic functions transform pipelines (materialize / update role / kind knowledge / data statements).

Browse / Input Dataflow: Context. REST HATEOAS state browse / render / submission. Interactive "dialogs". Form / Flow APIs.

Levels: contexts hierarchy polymorphism.

Metamodel:

Encoding. Context selectors (location / dataflow):

ID:Occurrence (CtxClass [metaclass, relationship context roles / Context CSPO Kinds. ID:Occurrence], TypeClass [class, relation roles. Match data with kinds. ID:Occurrence], TypeInst [instance, input / prompts roles facts / data event. ID:Occurrence]);

Resource : (ID: URL, Occurrence: Resource); Reified Resources, Roles, Contexts.

(Role, Occurrence, Attribute : Resource, Value : Resource) : Resource;

Relation / Statement (Resource / Relation, Resource / Role, Resource, Resource) : Resource; Relation instances TypeClass hierarchy (RoleRel, StatementRel, etc.).

Relationship (Role, Role, Role, Role) : Relation; CtxClass monadic wrappers hierarchy (RoleCtx, StatementCtx, etc.) wraps corresponding Relation hierarchy types.

<https://wiki.haskell.org/Zipper>

<https://wiki.haskell.org/Zipper_monad>

<http://learnyouahaskell.com/zippers>

<https://www.functionaljava.org/features.html>

<https://stackoverflow.com/questions/5919901/data-structure-differentiation-intuition-building>

<https://www.google.com/search?q=The+Derivative+of+a+Regular+Type+is+its+Type+of+One-Hole+Contexts&oq=The+Derivative+of+a+Regular+Type+is+its+Type+of+One-Hole+Contexts>

<https://github.com/jon-hanson/parsecj/blob/master/README.md>

<https://blogs.msdn.microsoft.com/lukeh/2007/08/19/monadic-parser-combinators-using-c-3-0/>

<https://www.w3.org/Data/events/data-ws-2019/>

<https://dzone.com/articles/functor-and-monad-examples-in-plain-java>

<https://dzone.com/articles/whats-wrong-java-8-part-iv>

<https://curiosity-driven.org/monads-in-javascript>

<https://hackernoon.com/functional-javascript-functors-monads-and-promises-679ce2ab8abe>

<https://medium.com/javascript-scene/javascript-monads-made-simple-7856be57bfe8>

<https://importantshock.wordpress.com/2009/01/18/jquery-is-a-monad/amp/>

<https://blogs.msdn.microsoft.com/lukeh/2007/08/19/monadic-parser-combinators-using-c-3-0/>

<https://github.com/jon-hanson/parsecj/blob/master/README.md>

<https://dzone.com/articles/parsing-in-java-part-3-diving-into-peg-parsers>

**MetaModel.docx**

Meta Model:

Arrangement of layered typed CSPO quads statements in which each CSPO role / type plays the role of "aggregating" previous layer abstract knowledge into more concrete aggregated contexts (statements) instances until a "reference model" CSPO type arrangement is achieved, which is the root of all other layers hierarchy.

The purpose of this is to achieve some ontology matching capabilities over an upper abstraction set of layers and to enable a Functional Knowledge Base Interaction APIs integration / virtualization overlay for matching and consumption of distributed datasets via endpoints dataflows.

**Reference Model:**

ID : URL;

Occurrence: Context;

Context : ID (Context / ID : intension, Object / Occurrence, Sign / Kind / Metaclass / Attribute, Value / Role / Class : extension);

Root of Meta Model hierarchy.

**Notation:**

[LayerType] : [LayerSuperType] ([ContextType], [SubjectType], [PredicateType], [ObjectType]);

**Layers:**

(Relation, Statement, Kind, Resource);

Resource : Context (Resource, Resource, Resource, Resource);

Statement : Resource (Statement: c, Resource, Resource, Resource);

Role / Class : Statement (Role: b, Statement, Resource / Attribute, Resource / Value);

Role / Class aggregating CSPO Resource (IDs) sharing Attributes for their Objects / Values.

Kind / Metaclass : Role (Kind: a, Role, Statement, Resource);

Kind: Aggregated similar Roles occurring as Resources (Object) in Statements (Predicate).

Relation / Entity : Kind (Relation, Kind, Role, Statement: c);

Rel type (Relationship) instance / bindings. An Entity (Relation: intension) and their Statements for its Kind / Role occurrences (occurrences: kinds / roles Relation plays in statements. Matching. Object: extension). Data (DCI)[1].

Mapping : Relation (Mapping, Relation, Kind, Role: b);

Rel players types / bindings scenarios. Information. Interaction (DCI)[1]. Mapping Role and Relation Kind: dataflow promoted types / order: relationships players domain / range. Entity alignment.

Relationship : Mapping (Relationship, Mapping, Relation, Kind: a);

Rel type declaration, player types. Knowledge. Context (DCI)[1].

Mapping and Relationship layer contexts are "calculated" (reifying) by Relation layer context kinds.

Semiotic Layer (ontology matching):

Value (Value, Value, Value, Value);

Sign : Value (Sign, Value, Value, Value);

Object : Sign (Object, Sign, Value, Value);

Context : Object (Context, Object, Sign, Value);

Reference Model:

Root of MetaModel hierarchy.

**Matching / Relations / Attributes:**

One of the intentions of having all this layered infraestructure is to be able to inspect "relations", being them "reified" into a Relationship construct, or being them single attributes and values for a subject enabling the possibility of "align" one into another for ontology matching purposes.

(a, b, c: Kind, Role, Statement): Reified Rel. to / from expanded Attributes / Values. Matching / roles (intension / extension).

Context DOM: parent / child; previous / next siblings; attribute / value (determined by CSPO roles). Class / instance DOM relation for parent / children layers instances.

**Ontologies:**

Context layers instances. Levels. Example: Dimensional ontology. Ontologies should be able to be built upon Reference Model layer CSPO types arrangements.

Dimension, Unit, Measure, Value.

Axis, Behavior, Flow (state change), etc.

Primitives: dimensional upper ontology. In / Out, Prev / Next, Pick / Drop, etc. Opposites. State change (current). Events, state flows. Marriage example.

**Message Dataflow:**

Relationship, Mapping, Relation streams / signatures. Messages: Context instances. Functional Knowledge Base Interaction APIs.

Aggregation: Browse / Transform.

Alignment: Inference.

Activation: Dataflow type (signatures). Message dispatch (domain / range ordered). Aggregation.

**Relations:**

Inference. Relation types: transitive, reflexive, simetric. Campo, alcance, dominio, rango, transform / function: infer / aggregate. Context functor / monad.

Inputs: (Context / Relation, PK, column, value);

Inputs: (Infer S Kind / Role, S, P, O);

Inputs: aggregate occurrences. Statement Context for each SPO as Occurrence with corresponding Attribute / Value (S: PO, O: SP, P: SO, etc.).

Inputs (infer rels): Part / Whole. SPO / OPS. Attribute / Value.

Inputs (infer rels): Containment. SPO / SPO. Parent / Children. Occurrences of Contexts of same Context layers (inherited Contexts). Example: (Mapping, Mapping) for (Relationship, Mapping). Super / Sub type Contexts instances relationships.

Inputs (infer rels): Order. SO Ps Domain / Range.

Input (infer rels): Event. Prev / Next state change. Type promotion.

**Reactive Functional Reified Metamodel:**

Transforms: Match Selectors. Hierarchy polymorphism. Contexts streams. Browse Metamodel. Context, Subject Selectors.

Transforms: Templates. Context instances (CSs) declaration / augmentation (POs) Selectors. Metamodel activation. Predicate / Object Selectors.

Encode Match / Template as Context. CS: Match, PO: Template Selectors. Apply Templates (role bindings / prompts) declaration / augmentation activation to matching selected CS streams.

Augmentation: Aggregation, Alignment, Activation Reified Match / Template dataflows. Reactive Model instances Match / Template dataflows.

Selectors:

Apply Role to Statement : Statement / Statement to Role : Role.

Apply Kind to Role : Role / Role to Kind : Kind.

Apply Relation to Kind : Kind / Kind to Relation : Relation.

Apply Mapping to Relation : Relation / Relation to Mapping : Mapping.

Apply Relationship to Mapping : Mapping / Mapping to Relationship : Relationship.

Apply Context to Relationship : Relationship / Relationship to Context : Context.

**Meta Model / Backend:**

Reference Model / Occurrences annotations matrix:

Attachment: show an example of a fully expanded set of last model layer (Relation) and its corresponding occurrences property graph annotations. Previous model layers may be annotated accordingly in respect to their ability to aggregate more abstract contexts (Resource, Context) properties.

Reference Model / Occurrences annotations matrix:

Lattice / FCA: Contexts / Resources. Objects / Attributes (Contexts instances axes). (X, Y): Z (for corresponding pair types / functional transforms).

Statements and annotations: FCA Lattice / FCA Contexts / Attributes. Objects / Attributes (Contexts instances axes). (X, Y): Z (for corresponding pair types / functional transforms). Typed calculus in FCA context development. Sets.

IDs / bitstring encodings. Algebraic / arithmetic activation flows / templates / transforms metadata encoded selectors. Vector Space Model quads polygon embeddings.

Interfaces:

ID : URL;

Context: ID;

Object : Context;

Sign : Object;

Value : Sign;

Labeled Property Graph annotations example. Augments Reference Model. Statement example, Statement context aggregates SPO annotations (Statement occurrences data in other Reference Model layers contexts):

Context (Context : Object, Object : Sign, Sign : Value, Value);

Resource : Value (Resource, Resource, Resource, Resource);

Statement : Resource (Statement, Resource, Resource, Resource);

Role : Statement (Statement, Role, Resource, Resource);

Kind : Role (Statement, Role, Kind, Resource);

Relation : Kind (Statement, Role, Kind, Relation);

Relationship : Relation (Relationship, Role, Kind, Relation); \*

\*: Relationship: Aggregated Relation Statement Relation (Object) Roles / Kinds.

Hierarchy: render / process layers (Relation, i.e.) as Context, Resource, Statement, Role, etc. (upper layers) contexts (i.e.: reify Relation as Context, Resource, Statement, Role, Kind layer contexts). Context semiotic layer: aggregation, ontology matching.

Intension / Extension: S / O.

sub / super hiers, containment: P / O.

Relation reification: Relation statement object: relation instance. a: Role / b: Kind: relation ends. (a): Role reifying rel attrs / values. (b): Kind Resource reifying rel subject (rel players).

**Model:**

Message Events Bus.

Context Monad & type hierarchy (AST). CSPO parameterized types & aggregation (layers hierarchies specializations).

DOM:

Layer (Contexts): events producer / consumer (streams observer / observable).

DOM Parsing MetaModel: aggregate occurrences containment / hierarchies. Layers: subtype / supertype browse parent / children / siblings (order) and Attributes / Values relations.

Context::matchFilter(arg : Context): signatures / kinds stream predicate.

Context::applyMap(arg : Context): apply updates (CS Contexts / PO CRUD) matching filter predicate Context. Fires event bus messages.

Matching applies to meta-model signatures (internal aggregation, alignment and activation augmentations) and to domain / actual models signatures. Render new Attributes / Values and CSPO statements.

DDD: Declarative AST / Dataflow VM. Runtime. Encodings (layers / messages / activations). Event sourcing backends (Blockchain). Patterns (DCI).

RDFS / OWL / Graph Backend: Reified Metamodel. Labeled Property Graph. APIs.

Functional Meta Model Context Layers. Reactive streams. Match / Template Selectors. API.

Stream Resources Connectors (P2P Connector Bundles Context I/O). Dataflow. Backends / Augmentations / Endpoints model layers. APIs (reactive / events).

Layered abstraction levels streams options / menu semantics (REST). Forms / Flows. Browse, match / transforms high level APIs. HATEOAS CRUD / Flows.

Patterns. Input formats. Sample data.

**FCA / Concept Lattices**

Model context statements of an upper ontology and occurrence statements for each ontology concept occurrences using FCA (Formal Concept Analysis) and rules from a pseudo-grammar.

Context statements / Occurrence statements: Grammars. Concepts / objects hierarchies: CSPO statements concept types / kind rules / terminal instances. Productions: concept types / kind rules / terminal instances mappings / flows.

Aggregate kind rules (grammar) into context statements / Occurrence statements: polymorphically, Kind context applies to all Resource hierarchy (all lattices).

Contexts parsing: monadic parser combinators / monadic AST. Recognize context types from (surrounding) reified kind types / rules (link grammar). TBD.

Rules are of the form:

(TypeA, TypeB) > AggregatedKindResources;

For example, in Relation lattice:

(someRoleA, someKindB) > AggregatedRelationResources;

Dispatch: Model (lattices) observer / observable (streams / functional) of grammar (contexts case match) events.

Lattice (FCA Contexts) population and Augmentation:

Base layer: Context. Resource, Statement, Role, Kind, Relation layers.

Input layer: case match layer grammar type signature. Reactive dataflow dispatch inputs top-down / bottom-up contexts layers hierarchy (lower hierarchy layers polymorphically materializes upper hierarchy layers) for Augmentation.

Lower hierarchy layers contexts mapping / function transforms into next upper layer context (example: Roles to Kinds). TBD.

Contexts (TBD):

Data layer: Resource, Statement, Role, Kind, Relation contexts (aggregation).

Information layer (occurrences / interactions): Data layer contexts statements products as new contexts (aggregation).

Knowledge layer (Dimensional / DCI Contexts): Information layer contexts statements products as new contexts (aggregation).

Tensor like arrangements by FCA / grammars (TBD):

Aggregations: Data layer contexts containing Information layer contexts containing Knowledge layer contexts. Encoding: embeddings, object / attribute features bitstrings, VSM (Vector Space Model) quad polygon angles. Order encoding (lattice). Activation / Aggregation (CUD): preserve order (lattice), arrange input message as production of corresponding context shape. Retrieval / transforms: resolve production of corresponding context shape for input message.

Augmentation:

Activation: Layer receives matching context message.

Aggregation: Layer resolves productions of grammar rules for context message.

Alignment: Resolution of relevant knowledge and input message. Emits output message.

Notes:

Context occurrences statements (i.e.: Statement in Kind context). Occurrence contexts: (S, S); (P, P); (O, O);

FCA: Lattice. Ordered Sets. Intension / Extension.

To Do:

Model Order. Axes: dimensions, units, measures. Events (measures / relations / state boundaries in data / information / knowledge levels: price, price at moment in time, variation, tendence predictions, idem for distances, etc.).

Hierarchies: metaclass / role, class, instance, occurrence (parent, children, previous, next, attribute, value). Encoding. Comparisons. Functional traversal (streams).

Dimensional contexts: Contexts from Occurrences contexts statements. Dimensional contexts: Events (attributes). Order relations assertions by context occurrences hierarchy domain / range, set / superset attributes relations.

(Mapping, Kind, Role, Statement);

Event (Dimensional context attributes): (Mapping / unit / class, Mapping super / parent / dimension / metaclass, Kind unit / measure / occurrence, Role measure / value instance);

Model Application domains: upper ontology (Behavior, Flow, etc.) encoded in meta model and specialization levels for domains contexts. Declarative abstractions ontology for application design: discovery, alignment and matching for services renderings and integrations.

Hierarchy: render / process layers (Relation, i.e.) as Resource, Statement, Role, etc. (upper layers) contexts (i.e.: reify Relation as Resource, Statement, Role, Kind layer contexts).

Context (reified / instances) kinds (topics): Resource content types. Resource (monad): representation, HATEOAS, dialogs. Dataflows (order / domain / range kinds).

HATEOAS: Form / Flow. Operations / Dataflow Representation / State IO (CRUD) prototypes / templates. Prompts (values / operations). Dialog. Gestures. Context: navigation state (i.e.: pick operation value prompt shows value type Form). DDD DOM.

**Meta Model (DCI)**

DCI Meta Models:

Data (data): Relation hierarchy Model.

Context (schema / dataflow): Augmentation hierarchy Model. Model layers extends corresponding Data layers.

Interaction (behavior / services) Dialog hierarchy Model. Model layers extends corresponding Data layers.

Contexts and Interactions Models extending / reified as Data Model layers enabling matching, inference and augmentations (FCA / ML embeddings for example) for behavior and schema alignments.

**Meta Model (Data)**

**Reference Model:**

ID : URL;

Occurrence: Context;

Context : ID (Context / ID : intension, Object / Occurrence, Sign / Kind / Metaclass / Attribute, Value / Role / Class : extension);

Root of Meta Model hierarchy.

**Notation:**

[LayerType] : [LayerSuperType] ([ContextType], [SubjectType], [PredicateType], [ObjectType]);

**Layers:**

Resource : Context (Resource, Resource, Resource, Resource);

Statement : Resource (Statement: c, Resource, Resource, Resource);

Role / Class : Statement (Role: b, Statement, Resource / Attribute, Resource / Value);

Role / Class aggregating CSPO Resource (IDs) sharing Attributes for their Objects / Values.

Kind / Metaclass : Role (Kind: a, Role, Statement, Resource);

Kind: Aggregated similar Roles occurring as Resources (Object) in Statements (Predicate).

Relation / Entity : Kind (Relation, Kind, Role, Statement: c);

Rel type (Relationship) instance / bindings. An Entity (Relation: intension) and their Statements for its Kind / Role occurrences (occurrences: kinds / roles Relation plays in statements. Matching. Object: extension). Data (DCI)[1].

Mapping : Relation (Mapping, Relation, Kind, Role: b);

Rel players types / bindings scenarios. Information. Interaction (DCI)[1]. Mapping Role and Relation Kind: dataflow promoted types / order: relationships players domain / range. Entity alignment.

Relationship : Mapping (Relationship, Mapping, Relation, Kind: a);

Rel type declaration, player types. Knowledge. Context (DCI)[1].

Mapping and Relationship layer contexts are "calculated" (reifying) by Relation layer context kinds.

Semiotic Layer (ontology matching):

Value (Value, Value, Value, Value);

Sign : Value (Sign, Value, Value, Value);

Object : Sign (Object, Sign, Value, Value);

Context : Object (Context, Object, Sign, Value);

Reference Model:

Root of MetaModel hierarchy.

**Dataflow Model (Context).**

Dataflow:

Iterations (types / kinds order declaration). Streams: Contexts / Occurrences Bus (signatures / discovery).

Conditionals (predicates / filters on types / kinds attributes / values) on Iterations.

Jumps (aggregation / stack sub-streams) on Conditionals. Apply Kinds on matching / referring Contexts (Employment, Person: Employee). Extract Kinds on matching / referring Contexts (Family, Father).

Order. Comparators: common upper hierarchies, Dataflow domain / range, SortedSet. Lattices (FCA contexts).

Augmentation: Aggregations, Alignments, Activations. Perform encoding dataflow.

Encoding: Augmentation, Template, Mapping, Transform.

OntResource: Ontology Matching (aligned URLs): semiotic context.

Dataflow Model:

Resource: (OntResource, OntResource, OntResource, OntResource);

Statement: (Transform, OntResource, OntResource, OntResource);

Role: (Mapping, Transform, OntResource, OntResource);

Kind: (Template, Mapping, Transform, OntResource);

Relation: (Augmentation, Template, Mapping, Transform);

Template: Reified Model and Model instances (hierarchies). Match inputs. Iterations.

Mapping: Dataflow reified operation flows bindings (subscriptions). Predicate / Object stream endpoints?. Conditionals.

Transform: Dataflow reified results. Jumps.

**IO / Services Meta Model (Interaction):**

Model for back ends synchronization and services exposures.

Context: (Model, OntResource, Resource, Resource); Key / value dictionary for source Model ontology matching.

Resource: (OntResource, OntResource, OntResource, OntResource); Full matched Resource descriptions: Type, ID, Attributes, Values.

Statement: (Assertion, OntResource, OntResource, OntResource); Transform (Jumps). Source Model(s) interaction interface Services URLs (IO).

Role: (Prompt, Assertion, OntResource, OntResource); Mapping (Conditionals). Predicates (LHS: Predicate, RHS: Object).

Kind: (Interaction, Prompt, Assertion, OntResource); Template (Iterations). Aggregate matching Interaction Assertion Prompts.

Relation: (Dialog, Interaction, Prompt, Assertion); Augmentation. State flows.

**FCA / Lattices**

For each layer context statement build tables which axes correspond to each context CSPO context types. Aggregate CSPO types / values in the form shown below.

FCA Context (tables): Context (Relation, Kind, Role, Statement, Context, etc.) matrices w./ corresponding CSPO x CSPO types axes. Types intersections determine cell type (as in example tables shown). Values intersections are instances of corresponding types. Example:

(RoleA x StatementB: KindC) : Kind RoleA plays in StatementB.

Thus, each layer context statements are used to build a matrix of CSPO x CSPO of its types and (scaled) values. In this manner (aggregating matrices / FCA contexts), SPO Resource occurrences in matrices axes SPOs / context layers statements SPOs are reified in statements for which the occurring SPO Resource is the statement Context and its SPOs are resolved according its SPO position in the original Context. This statements form the matrix rows and columns SPOs according some aggregation layout.

The purpose of this is to retrieve enough concepts (FCA) metadata to populate concepts / objects / attributes conforming a Lattice of related Resources and those relations values (as in the above example).

FCA Lattice (concepts / attributes / objects): (types / values) x (types / values). Encoding (IDs): ontology matching enabling type / instance calculations / traversal / transforms.

Relation matrix:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Relation | Kind | Role | Statement |
| Relation | Relation | Kind | Role | Statement |
| Kind | Kind | Relation | Statement | Role |
| Role | Role | Statement | Relation | Kind |
| Statement | Statement | Role | Kind | Relation |

Matrices for other layers (Kind, Role, Statement, Resource, Context) contexts follows the same principles.

Reifying one aggregated layer SPO layer (for example: Kind in the previous table) has original context matrix axes in the corresponding SPO layer (Subject in this case):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Kind | Relation | Statement | Role |
| Kind | Kind | Relation | Statement | Role |
| Relation | Relation | Kind | Role | Statement |
| Statement | Statement | Role | Kind | Relation |
| Role | Role | Statement | Relation | Kind |

The “generic” form of the table is:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Context | Occurrence | Attribute | Value |
| Context | Context | Occurrence | Attribute | Value |
| Occurrence | Occurrence | Context | Value | Attribute |
| Attribute | Attribute | Value | Context | Occurrence |
| Value | Value | Attribute | Occurrence | Context |

Use cases:

Use FCA Lattice for sorting / ontology matching / augmentations / query / ontology browsing.

Aggregation: Complete contexts objects / concepts / attributes by FCA / inference.

Inference example: (Statement x Statement): Relations between both Statements.

Learning: ML embeddings for types / values / concepts.

TBD: (metaclass, class, occurrence, instance) relations / atttributes.

TBD: Set oriented intension (C) / extension (O) and relations between sets.

TBD: Discover IDs / encoding techniques enabling algorithmic translation of models operations.

Encoding: FCA Scaling. FCA Context objects and attributes are corresponding CSPO Contexts types scaling enclosed Context types instances. A potential encoding of axes objects and attributes (rows and columns) would be a bitstring of length 4 x n, being n the length of an instance identifier for each quad Context encoded in its corresponding bitstring quad space (4 is for CSPO quad types instances identifiers segments). Then, navigation should be allowed from a pair of object / attribute to another object / attribute: (type, object) x (type, object): (type, object).

If Context types / instances identifiers are sequential in form, a mapping (hashing) could be done in a bitstring of the length of scaled attributes (columns) having a 1 in the corresponding attribute for a given object. Attributes also may be rendered as a sequence of prime numbers being an object extension the product of its attribute primes.

Layout: The aggregated statements have as Contexts the occurring SPOs in a Context layer statement and its SPOs are the occurrence Context and the other SPOs in the occurring statement. For a Context in an aggregated statement occurring as (SPO) in the occurrence statement, occurrence statement Context is its (SPO) and its aggregated (SPO) is occurrence statement (SPO).

Layout: Having a Context layer, a matrix (FCA context) of the form (CSPO x CSPO) is built for aggregation of models. The aggregated statements (rows / columns) have as Contexts the CSPOs (occurrences) of an axis and and its SPOs are given from the types / values of the context layout. For an aggregated Context statements / matrix, the original context is located in the (SPO) axis from the (SPO) which it was taken from the original Context matrix.

Encoding: Layout rules. Context layers, use layout to aggregate occurrences matrix. Use aggregated occurrences matrix to retrieve original matrix.

Encoding: Layout. Layers. Aggregation: upper layers / lower layers. Encode Augmentations. Browse. Transforms. Reified Model. DCI.

Layer declarations: TBD (Context, Occurence, Attribute, Value) reified types / data.

Resource matching (reified / data): context / occurrence rules. Context population. Types / Functional / Data Models.

Layer downwards: Layer for which Subject is Context.

Layer upwards: Layer for which Context is Subject.

Facets: Concept hierarchies common attributes. Types: reified model objects instances: contexts / layers / aggregations. Values. Parsing (case match): resolve if an (scaled) type / value object corresponds to an (scaled) type / value attribute and which is its intetsection type / value (grammars / signatures / aggregation / dataflow). TBD.

Facets: cells Context type: from corresponding Kinds for reified CSPO roles of SPO axes intersections (type intension). Context values: CSPO Resources of Kind type extension.

CSPO / Kinds types / values (FCA scaled types / values objects / attributes intersections): Multiset (OrderedSet) encoding of CSPO Statements and Kinds as bitstring quad (CSPO segments) for corresponding sets / elements: C, S, P, O Resources arranged in a three SPO sets distribution (diagram) with SP intersection for Object Kinds, PO intersection for Subject Kinds and OS intersection for Predicate Kinds (Kinds being referencing SPOs in their corresponding parts and aggregating Kind values in their Kind type part).

Kinds layouts:

(S, P): OK; (P, O): SK, (O, S): PK; (Idem, Idem): Contexts, i.e.: Relations of equivalent Context signatures.

CSPO Sets Context (data layer, context / interaction hierarchies reification) population: parsing over RDF quad statements IO. Create / aggregate quad bitstring Resource / Kind identifiers. Functional query / browse / transform. Populate FCA contexts / lattices and calculate cells / dataflow (sets streams).

Populate Contexts (Sets, FCA, layers): Resources, Statements: inputs.

Role (CSPO Context hierarchies reified roles), Statement, Resource, Attribute. From sets aggregation.

Kind, Role, Statement, Resource. From sets Kinds aggregation.

Relation, Kind, Role, Statement. From sets reified Kinds / Context / Statements aggregation (Kinds domain / range).

Sets Layout and encoding bitstring mask format.

**Items.docx**

**Objectives:**

Rosetta Stone like engine enabling Ontology (Data, Schema / Information, Knowledge / Behavior) discovery, matching and integration.

Reactive Service Bus for pluggable integration of application and translation of gestures between domains business systems allowing workflows alignment and discovery of application systems behavior.

**Layers: RDF Quads Representation. Augmentation / Inference Matrix Models**

Patterns:

(Context, Occurrence, Attribute, Value);

(Dimension, Measure, Unit, Value);

(Dimension, Resource : SPO, Kind, Statement);

Sets Model Layers Structure:

(Dimension, Resource, Kind, Statement);

(Statement, Dimension, Resource, Kind);

(Kind, Statement, Dimension, Resource);

(Resource, Kind, Statement, Dimension);

(Dimension, Resource, Kind, Statement);

Dimension: U

Resource: SPO

Kind: SPO Intersections (pairs)

Subject Kind: P intersection O

Predicate Kind: S intersection O

Object Kind: S intersection P

Statement: SPO Intersection (of the three sets)

Sets / Individuals Mappings:

IDs: metaclass, class, instance, context, role, occurrence, previous, next ID roles relations for Model Set Contexts.

Augmentations / Transforms: Model / Domains functional mappings. Order. Dimensions. Axes. Flows. Hierarchies. Inference / Population.

Levels: Augmentations. Mappings.

Levels: Resource, Kind, Statement.

Levels: Reify Statement as Kind, Kind as Resource, Resource as Statement.

Levels: Reify Resource as Kind, Kind as Statement, Statement as Resource.

Sets / Individuals Mappings:

Levels (layer statements) shifts (quads matrix). CSPO roles:

(Dimension, Resource, Kind, Statement);

Sets / Individuals Mappings:

Levels (layer statements) shifts (quads matrix). CSPO roles:

(Dimension, Resource, Kind, Statement);

Resource: Statement Semantic Address / ID / Occurrence. Dereference / browse: Representation State / Roles (CSPO) according request / response referrer / occurrence / transforms / mappings browsing context states. Representation exchanges: Forms / Flows. Examples: Faceted search (build context template schema roles via form request / response exchange updates), Shopping Cart (build use case representation items via state representation request / response exchange updates). Representational state transfer on each request / response bodies. Placeholders / Flows history Roles. Stateless services: product representation submit to order service submit to payment service submit to delivery service. Functional dataflow, distributed representation services (URNs).

Encoding. Index Lists: C ID of SPOs Statements. CS ID of POs Statements. CSP ID of Os Statements. Schema: Zippers. Representation: cons cells. Functional Mappings: data types / monads.

Augmentations (purposes) specification encodings discovery: Self organizing graphs / maps. Fitness function. Problem (encoded / dataflow signatures) solver.

IDs:

Data Schema Representations UIDS. Results of Behavior Interactions. Messages Dataflow Flows Blockchain. Contextual ID: State of Entities at Representation Dimensional State Context (state respect to other entities dimensional state in various axis).

Example: Quads Context State Representation (dimensions state snapshot) ID calculated from (CSPO Subject example):

CSPO Subject parents state: CSPO Contexts.

CSPO Subject children state: CSPO Predicates.

Idem for other CSPOs. Encode, address / serialize in such a way that allows retrieval of entity ids, state: data, schema: roles, behaviors: available mappings / transforms state by dimensional contextual state spaces of an entity. Example: Tell me the weather conditions of the day of the playing of the match between A and B and B wins the game.

IDs:

Relations, Relationships, Zippers parse / materialize.

Upper / dimensional knowledge. Apply / classify Patterns. Relations encoding / sets quads. Functional Transforms: Functor(Role, Definition Transform) : Functor(Role).

IDs encoding. Functional mappings / roles. Patterns: definitions / instances. Matching definitions materialize instances. Relation types / Relation relations (inverse, complement, etc.).

Dataflow: FamilyMember<Son> (FamilyMember<?> -> FamilyMember<Uncle>) Transforms I/O Flows.

Augmentations: materialize mappings. Metaclass, class, instance, context, occurrence roles. Functor mappings / transforms contexts / arguments (dataflow): roles.

Examples: Uncle. Marital status. Temporal. Containment. Order. Time. Hierarchy. Containment / Order: Graphs. Trees. Lattices.

IDs:

Content Addressable RDF (Memory / Network). Dataflow (executions / query transforms flows: next address activation).

MetaModel: Content Addressable Network Nodes: Quads matrices projected in a virtual multi-dimensional Cartesian coordinate space, overlay network, on a multi-torus?.

Jena SPARQL Services (code / functions) CSPO Resources. Contextual (parameterized states) specification (vars, wildcards, placeholders) of statements expansion.

Services: W3C DIDs Contracts. Rules. Application: StratML / Integration. Logs / Workflows / BPM / Dashboards.

Encoding: Functional / DOM / OGM:

Content Addressable: Context / Content IDs. Pseudo grammar:

Locator: URL / URL + OccurrenceContext;

OccurrenceContext: Locator(Locator) + Locator;

SPO: Locator + OccurrenceContext;

ID: SPO / SPO + OccurrenceContent;

OccurrenceContent: ID(ID) + ID;

Resource: ID + OccurrenceContent;

Node: Resource / Resource + Arc;

Arc: Node(Node) + Node;

Context: Node + Arc;

SPO: Resources.

Resource: Kinds.

Context: Statements.

Materialize Quads layers hierarchy roles. Functional API (TBD). S-Expressions. Forms.

Quads layers hierarchy roles:

(Dimension, Resource, Kind, Statement);

(Dimension, Measure, Unit, Value);

(Context > Occurrence > Attribute > Value);

C: Functor Wrapper Type.

S: Functor Value Type.

P: Functor Mapping / Transform.

O: Functor Transform Wrapped Values.

Functional Functors / Transforms roles:

Resource Statements:

C: Metaclass. Context.

S: Class. Subject.

P: Context. Predicate.

O: Instance. Objects.

Dimensional Statements (Mappings: order / dataflows) transforms:

C: Input Role / Kind. Context.

S: Occurrence. Resource. Subject.

P: Role. Output Kind. Predicate.

O: State. Statements. Object.

Functors / Mappings: Dimensionally Augmented Upper / Inferred domain knowledge. Resources Dataflow / Order populated Dimensional Mappings.

Apply Dimensional Transforms: Reactive materialization of matching Resource Kinds over Dimensional Mappings. Pipeline outputs with matching mappings outputs / inputs (order rules).

Transforms Specifications:

Upper Augmentation Dimensional Transforms: Activation, Alignment, Aggregation over core upper / root model roles.

Domain Transforms: declaratively stated in Resources and Dimensional models. Upper alignments. TBD.

Encoding: Quads Matrices / CAN (Content Addresable Network) coordinate space.

Axis: Quad Statements layers / projections. Dimensions / Functional Mappings:

(O, C) : (C, S) : (S, P) : (P, O)

(O, C) : Dimensional Context. Object Dimensional extension. Functional Mappings / Transforms context (object member property traversal).

C, S, P, O Axis Dimensional Mappings / Transforms populated from Augmented Quads CSPO Roles (axis / dimensions):

(Dimension, Measure / Resource, Unit / Kind, Value / Statement);

Addressing: Semantic hashing (context / content functional) references data (entities). CSPO Functional Roles IDs. Address space functional / traversal.

Hashing (content addressable) IDs: VSM (Vector Space Model). FCA (Formal Concept Analysis). Index / mappings / encodings. Render quad polygon / vectors similarity function (context / content polygon sides / angles).

Graph / ANN CAN CAM: Augmentations (align, activate, aggregate layers) functors / transforms. FP / ML. Papers (DGI Infomax).

Typed RDF, monads, bnodes (wrapper types, i.e: Address primitives / locators / bnodes fields. Drafts. Dimensional: quads: sets / elements / transforms definitions / declarations. Relationship / Relation higher level ontology / objects.

Augmentation: Graph / ANN FP / ML Data flows. Nodes: functors. Activation functions: transforms (meta model roles / meta relations lambdas: metaclass, class, etc.). Data flows: Quads input patterns activation, alignment, aggregation transforms pipelines results according graph context / state.

Property Graphs: CSPO Network Occurrences Attributes (metaclass, class, context, instance, role, etc.) of Graphs Nodes inputs / outputs. State. Example: Context for which Predicate Occurrence is valid. Order: hierarchies.

Domains Graph:

(C, S, P, O);

(Relationship, C, S, P);

(Relation, Relationship, C, S);

(Entity, Relation, Relationship, C);

Relationship: Declarative aggregation of instance types (Relations) contexts, subjects, predicates (attributes). Marriage.

Relation: Aggregation of Relationship instances role types declarations. Wife, Husband, Wedding. Type attributes due Relationship aggregated predicates.

Entity: Aggregation of Relation contexts role instances. aWife, aHusband, aWedding. Attribute values due Relation Relationship aggregated predicate objects.

Layout:

(Context, Occurrence, Attribute, Value);

Relationship:

Marriage Context: One Context Statement for each Marriage Relationship Predicate: all Marriage(s) Relationship statements. Aggregate Relationships types.

Relation:

Wedding Context: One Context Statement for each Wedding Relation Subject. Relation is a Relationship (metaclass / occurrence) instances aggregation: all Marriage Relationship Wedding(s) statements. Aggregate Relation types (metaclass).

Entity:

aWedding Context: One Context Statement for each aWedding Entity Context. Entity is a Relation (metaclass / occurrence) instances aggregation: all Wedding Relation aWedding statements. Aggregate Entity types (Relations) instances.

C, S, P, O: Meta Model metaclass, class, instance, context, role, etc.

Dimensional / CSPO Alignments.

Dimensional Matrix:

(Kind, Relationship, Relation, Entity);

(Entity, Kind, Relationship, Relation);

(Relation, Entity, Kind, Relationship);

(Relationship, Relation, Entity, Kind);

Kind: Type inference. Hierarchies, Context sets statements / attributes declaration. Aggregate Relationships types. Aggregate Relations (metaclass) types. Aggregate Entity types (Relation) instances.

DGI (Deep Graph Infomax) / CAN Tensors / Nodes / Dimensions.

Dimensional Matrix Graphs / Network Inferences:

Augment SPO input statements into Relationships hierarchy matrices aggregated quads. Example: (:Peter :wife :Mary) entails a Marriage, a Wife, a Husband, a Wedding, aWife, aHusband, aWedding layers contexts instances / attributes for input statement augmented via dimensional matrices instances kinds / links / matching (activation, alignment, aggregation) on previous knowledge. Prompt and augment funtional shapes bindings layout (CSPO Dimensional / semiotic context / metaclass, class, instance, context, role, occurrence) matching signatures: Data flow.

Expand all Relationship hierarchy matrix aggregated quads knowledge back to SPO statements. Relationships / SPOs reactive updates: Data flow expressions for further knowledge updates (CSPO Dimensional / semiotic context / metaclass, class, instance, context, role, occurrence) shapes bindings layout. Statements / Relationships functional concepts slots (variables / signatures reactive placeholders). Produce and augment new matching knowledge statements.

RDF Graph Literals: Contexts. Bnodes: lambdas. OWL Facets, SPARQL Functions, RIF Operations.

Monads Meta Model:

Functor type: Resource<? extends URI>.

Wrapped types hierarchy (Contexts):

(URI, Statement, Kind, Occurrence);

(Occurrence, URI, Statement, Kind);

(Kind, Occurrence, URI, Statement);

(Statement, Kind, Occurrence, URI);

Transforms:

Core Transforms: context, metaclass, class, instance, occurrence, role Resource functions. Browse Resource graph contexts / occurrences / relations / attributes. Transforms yields meta model contexts Resource roles. Navigate data / schema / behavior.

Transform pipelines: state context traversal (domain / range). Example: Person, Address, Street, Number. GetAddress(Person / Address) transform, GetStreet(Address, Street) transfom, GetNumber(Street, Number) transform.

Transform Dimension Models Hierarchies:

Semiotic Dimension. Core Data Model I/O, CRUD / persistence events. Meta Model templates dispatch / dataflow to corresponding dimensions:

(Value, Context, Object, Attribute);

(Attribute, Value, Context, Object);

(Object, Attribute, Value, Context);

(Context, Object, Attribute, Value);

Inputs / Outputs events (sync) are of the shape: (TableName, PKVal, ColName, CellVal);

provided / consumed by any "tabular"-able serialization format (XML, JSON, etc.).

Semiotic (Sets) Model / Mappings:

(Context / Set / Kind, Occurrence / CSPO Individual, Attribute, Value);

Mappings / Kinds Aggregation:

SPO Kinds: SPO Context typed Kind.

SPO Kinds : Implements generic Kind (Models).

Aggregate Kind Context Mappings for all Sets Model SPO statements.

(SubjectKind, Subject, PredicateKind, Predicate);

(SubjectKind, Subject, ObjectKind, Object);

(PredicateKind, Predicate, SubjectKind, Subject);

(PredicateKind, Predicate, ObjectKind, Object);

(ObjectKind, Object, SubjectKind, Subject);

(ObjectKind, Object, PredicateKind, Predicate);

Hierarchies / Order:

Statement Predicate as Subject, Object as Predicate.

Dimensional: Measure as Dimension, Value as Unit.

Meta Model Dimension. Normalized representations of aggregated Dimensions models data / metadata:

(URI, Statement, Occurrence, Kind);

(Kind, URI, Statement, Occurrence);

(Occurrence, Kind, URI, Statement);

(Statement, Occurrence, Kind, URI);

Context : Statement;

Object : Occurrence;

Attribute : Kind;

Value : URI;

Relationship Dimension (Domains): Aggregated domain knowledge ontologies of Meta Model / Semiotic populated layers (upper reactive dataflows).

(Entity, Relationship, Relation, Kind);

(Kind, Entity, Relationship, Relation);

(Relation, Kind, Entity, Relationship);

(Relationship, Relation, Kind, Entity);

Relationship : Statement;

Relation : Occurrence;

Kind: Kind;

Entity : URI;

Dimensional Dimension: Cube like aggregation of Meta Model / Domains / Semiotic populated layers (upper ontologies reactive dataflows).

(Value, Dimension, Measure, Unit);

(Unit, Value, Dimension, Measure);

(Measure, Unit, Value, Dimension);

(Dimension, Measure, Unit, Value);

Dimension : Statement;

Measure : Occurrence;

Unit : Kind;

Value : URI;

Dimensional Interoperation: Functional Mappings and Models schema class hierarchies. Resource Monad. Semiotic Value and Domain Entity sharing the same identifier: Framework URI (OntURIs).

Dimensional hierarchies: Functional Mappings and data flow for reactive base upper ontology templates:

Dimensional population:

Contexts class hierarchy plus reactive functional dataflow in upper ontology shapes / templates. Resolve missing Resources or they placeholders (functional transforms APIs): data flow Values.

Transform Templates / Augmentation:

Mappings / Transforms:

(DomainSelector : Kind, Input : URI, RangeSelector : Kind, Value : URI);

Multiple Values.

Nested Transforms (Value Input context).

Assert Transforms: Materialize Meta Model assertions.

Query Transforms: Values occurrences (contexts) for Meta Model Domain Inputs.

Mapping signatures: Kinds I/O. Shapes. Templates. DOM.

Inputs / Values: Reactive data flow slots.

Shape (Transform instance) compliance: Input / Value slots fulfilment (Value / Object domain).

Template (Transform spec) layout rendering: Dimensional layout Inputs / Values (Context / Subject domain).

DOM: Functional Dynamic Object Model.

Transforms: Streams. Emits Contexts (from Aggregated layers Objects), emits Contexts Subjects, Subjects Predicates, Predicate Objects aggregated into Statements.

Activates / Aligns into Dimensions. Activate types: perform CSPO Occurrences Kinds activation. Align layers: aggregate layer context instances occurrences. Flows Selector Predicates.

Dimensions layers (matrices) functional data flow pipelines. Contexts, occurrences, attributes, values templates (data flow Resource: matching kinds) reactive resolution for definitions of instances (upper ontology / data flow slots).

Domain Transform: Kind wrapped Resource. Dataflow resolution: Kinds I/O signatures (domain / range). Domains Transforms (lambdas / graph template resources). Navigate domain (types / instances / attributes / relations).

**Refactor: TBD:**

Kinds are instances of Types in a Role in an Occurrence of a Context.

Type: Aggregated Context Occurrence SPO. Sets.

Kind: Aggregated Kind Occurrence SPO Context augmented Role Type occurrences. Type reification. Statement reification: property graphs.

(Kind, Context, Role / Type, Occurrence);

Matrices: (X, Y) invocation of Functors over Resources. Resource Mappings.

To Do.

Model:

Context: CSPO Contexts Occurrences.

(Type, Context, Attribute, Value);

(Kind, Type, Context, Attribute);

Mappings (Models):

(Kind, Context, Type, Context);

(Type, Context, Kind, Context);

Mappings:

CSPOs.

Metaclass example: Type of a Kind (Developer kindOf Employee).

C: Type.

S: Context.

P: Kind.

O: Context.

(Statement, Type, Kind, Context);

(Type, Kind, Statement, Context);

Kind: Context Resource Statement Type Reification. Type Kinds in model statements (type occurring). Statement reification: property graphs.

(Relationship, Relation, Kind, Entity);

(Dimension, Measure, Unit, Value);

Order / Hierarchies: S as C, O as P. Dimensional example: To do.

APIs: Models. Functional. Classes. Monads, Transforms. I/O, CRUD.

Normalize Models:

Context: CSPO Contexts Occurrences (Model hierarchies). Resource Monad containing role instance.

Metaclass example: Type of a Kind (Developer kindOf Employee).

Core Meta Model:

(Type, SubjectContext, Statement, Kind);

SubjectContext: for a given Type, stream of its aggregated statement same subject role occurrences. (subject attributes). Idem for other statement roles data flow population.

(Kind, ReifiedType, SubjectContext, Statement);

Kind: Augments Type instance with contextual role semantic attributes.

(Statement, Kind, ReifiedType, SubjectContext);

(SubjectContext, Statement, Kind, ReifiedType);

Hierarchy / Aggregations:

Context.

Statement : Context.

Kind : Statement.

Type : Kind.

Models:

Discrete events model:

(Relationship, Relation, Kind, Entity);

Hierarchy / Aggregations:

Relationship : Context.

Relation : Relationship, Statement.

Kind : Relation, Kind.

Entity : Kind, Type.

Continuous measures model:

(Dimension, Measure, Unit, Value);

Hierarchy / Aggregations:

Dimension : Context

Measure : Dimension, Statement.

Unit : Measure, Kind.

Value : Unit, Type.

Normalize Mappings:

Mappings: R / describe. Transforms (CUD): Mappings / Transforms Models layout. Functional invocation (streams / signatures): schema / domains matching / execution.

Transforms Resource layout: Types (including reified roles, i.e.: Relationship Resource instance). CUD.

Mappings Resource layout: Kinds (including reified roles, i.e.: Entity Resource instance). R.

Matching Resource layout: Statements.

Output Resource layout: Statements (of augmented instances of Types / Kinds).

Core Model reifies models (events / measures) CSPO statements for augmentation and inference. Core Transforms / Mappings for Models Augmentation / Population (reified roles).

To do:

Normalize / homologate models / mappings / transforms. Functional APIs / reactive streams: I/O: Types / Kinds Aggregation. Mappings / Transforms Models Augmentation / synchronization.

Mappings / Transforms:

Mappings: retrieve / browse graph.

Transforms: CUD graph manipulation.

Context Functional layout: Input: Contexts. Output: Statement stream.

Statement Functional layout: Input: Statements. Output: Kind stream.

Kind Functional layout: Input: Kinds. Output: Type stream.

Type Functional layout: Input: Types. Output: Subject stream.

Augmentation:

Resource Functional Matching:

Statement, Kind, Type, Subject class hierarchy.

Statement, Kind, Type, Subject instances hierarchy.

Input: Statement. Pattern. Selector.

Pattern CSPO Matching (traversal):

* CUD: (C, (C, S, (C, S, P, (C, S, P, O)))). If not exists, create / append (role factory). Transforms.
* R: (C, (C, S, (C, S, P, (C, S, P, O)))). If not exists, retrieve last. Traversal.

Predicate exists: data flow streams ontology matching. Append: align models (references). Augmentations.

Output: Augmented Statements.

Models Mappings:

Discrete events model:

(Relationship, Relation, Kind, Entity);

Continuous measures model:

(Dimension, Measure, Unit, Value);

Meta Model Reification (sync):

(Context, Statement, Kind, ReifiedType);

CSPO models Resource classes: meta model classes inheritance.

Events domains, dimensional model, core: upper ontology (primitives) for functional matching data flow / models aggregation / layers. Order / hierarchies. Dimensional / events core classes.

**Runtime / Component:**

Input Statements:

Wrap aligned input Statement Resources into Resource monads. Populate functional Statement with Resource monads instances wrapping aligned source Resources.

Factories: To do. Current Model / Input Model.

Factory CSPO streams actions:

Init: Session. Current model / Object Model (models runtime / persistent state).

Factory Collectors / API patterns.

Dataflow: CSPO Builder slots (resolved / resolvable mappings / transforms).

Kind / Type: Activation (Context / Statement aggregation). Aggregate Kinds / Types incrementally into functionally resource resolved / resolvable Futures streams (dataflow builder slots).

Append / Delete (versions): Aggregation. Resolved Kinds / Types aggregation of Statements / Contexts functionally into resources resolved / resolvable: Futures streams (dataflow builder slots).

Update / Link (relations, i.e: sameAs, schemas: uncle): Alignment. Functional inference: models (metamodel, discrete, dimensional) ontology statements. Apply Mappings / Transforms. Dimensional / Events / Model schemas / quads translation / transforms.

Update Factory session runtime model.

Comparison: Order, Hierarchy. Loop for each input resources over a model resource of a model stream.

Builder: Aggregate Statements (Persistence / Factory Models).

Update model persistent state.

Builder: Aggregate Statements (Persistence / Factory Models):

Statements stream: Functional Model Contexts. Pairwise streaming merge of persisted model resources with Factory session state augmented resources. Emits streams of augmented resources (CSPO) / Statements: Persistence / Mappings / Transforms / Factories subscribers.

For each layer (Meta Model): Feed Input: Statement. Pattern. Selector. Infer matching input Resource types (Factories: functional Mapping / Transforms shapes / descriptions builder of model contexts via inputs aggregation / inferences).

Builder Pattern CSPO Matching (traversal):

* CUD: (C, (C, S, (C, S, P, (C, S, P, O)))). If not exists, create / append (role factory). Transforms.
* R: (C, (C, S, (C, S, P, (C, S, P, O)))). If not exists, retrieve last. Traversal.

Matching resolves CSPO next item in Transforms / Mappings model streams traversal contexts. Builder / Factory builds / update items upon its input Statement Resource (resolve types / contexts), functional context and model metadata. Resulting statements are appended / removed from the model.

Data flow traversal augmentation. Functional Resources / Contexts model flow roles traversal paired with inputs Resources / Contexts resolving augmented Resources (CUD / Merge).

Streams, Collector, Map Reduce. Data flow: CSPO aggregation. Parse / build CSPO model contexts / mappings / DOM. Factories streaming model context resources (chained CSPO streams) resolve new / existing contexts resources given input resources CSPO matching model resources streams CSPO augmented contexts (CUD).

APIs. XML:

XML / XSLT: XUL Application. Node: Augmentation Services Endpoint (I/O). Templates. Connectors. REST. DCI (XML code / data HAL I/O).

APIs. JavaScript:

JS / JSON / JSON-LD / Semantic / DOM / Functional QL (CRUD). TBD.

Input: RDF Quads.

ResourceRegistry: Mappings. Naming / Matching. Streams. Index. Functional Services.

Model Layers: StatementBuilder. Mappings.

Augment Models. Mappings. REST Context (referrer). Factory CSPO streams actions.

ResourceRegistry: Mappings. Naming / Matching. Streams. Index. Functional Services.

Model Layers: StatementBuilder. Mappings.

Output: RDF Quads (graph).

APIs: Facades. I/O Data CRUD Formats / Endpoints.

Inputs / Naming:

Input Quads: Statements from R2RQ, Apache Any23, Stanbol, MetaModel, Olingo, JBoss Teiid.

Resources Wrapper:

OntResource (Source URIs, Model URN);

Sources Alignment.

Contextual / Metadata URN:

CSPO URN IDs. DIDs.

Functional alignments (Mappings / Transforms / Layers).

Naming / Registry

IDs / Roles, Contexts:

Context Role Class (i.e: Dimension, Entity)

Context Role Class Instance

OntResource ID

CSPO ID

Metaclass

Class

Instance

Context

Occurrence

Role

urn:ont:[ID:ID][ID:ID][ID:ID]

URN IDs Syntax / Grammar.

Resolution: Traversal / Patterns / Predicates.

Blockchain: DHT CAN. Revisions: CUD / Versions.

Contracts: Mappings / Transforms.

Resources Monad<OntResource>.

IDs: Encoding (order / hierarchy). Aggregation. Dimensions. Bitstring lattices.

Patterns: Encoding Augmented with new knowledge still matching with previous one (Contexts, Subjects, etc.).

Augmented via Augmentation / functional data flow traversals aggregation (encoding).

Resource Monad

Statement Contexts

Wrapped Resource Type (Dimension, Unit, Measure)

Wrapped Resource Instance (Time, Minutes, 60)

Wrapped Resource Monad occurrence (Statement Context)

(TypeID, TypeInstanceID, WrappedResourceID, WrappedResourceOccurrenceID);

Statement resolves / matches IDs. Futures. Patterns, hierarchies, order resolution of Resources (data flow traversal build / match).

Input Statements:

Wrap aligned input Statement Resources into Resource monads. Populate functional Statement with Resource monads instances wrapping aligned source Resources.

Factories: To do. Current Model / Input Model.

Factory CSPO streams actions:

Init: Session. Current model / Object Model (models runtime / persistent state).

Factory Collectors / API patterns.

Dataflow: CSPO Builder slots (resolved / resolvable mappings / transforms).

Kind / Type: Activation (Context / Statement aggregation). Aggregate Kinds / Types incrementally into functionally resource resolved / resolvable Futures streams (dataflow builder slots).

Append / Delete (versions): Aggregation. Resolved Kinds / Types aggregation of Statements / Contexts functionally into resources resolved / resolvable: Futures streams (dataflow builder slots).

Update / Link (relations, i.e: sameAs, schemas: uncle): Alignment. Functional inference: models (metamodel, discrete, dimensional) ontology statements. Apply Mappings / Transforms. Dimensional / Events / Model schemas / quads translation / transforms.

Update Factory session runtime model.

Comparison: Order, Hierarchy. Loop for each input resources over a model resource of a model stream.

Builder: Aggregate Statements (Persistence / Factory Models).

Update model persistent state.

Builder: Aggregate Statements (Persistence / Factory Models):

Statements stream: Functional Model Contexts. Pairwise streaming merge of persisted model resources with Factory session state augmented resources. Emits streams of augmented resources (CSPO) / Statements: Persistence / Mappings / Transforms / Factories subscribers.

For each layer (Meta Model): Feed Input: Statement. Pattern. Selector. Infer matching input Resource types (Factories: functional Mapping / Transforms shapes / descriptions builder of model contexts via inputs aggregation / inferences).

Builder Pattern CSPO Matching (traversal):

* CUD: (C, (C, S, (C, S, P, (C, S, P, O)))). If not exists, create / append (role factory). Transforms.
* R: (C, (C, S, (C, S, P, (C, S, P, O)))). If not exists, retrieve last. Traversal.

Matching resolves CSPO next item in Transforms / Mappings model streams traversal contexts. Builder / Factory builds / update items upon its input Statement Resource (resolve types / contexts), functional context and model metadata. Resulting statements are appended / removed from the model.

Data flow traversal augmentation. Functional Resources / Contexts model flow roles traversal paired with inputs Resources / Contexts resolving augmented Resources (CUD / Merge).

Streams, Collector, Map Reduce. Data flow: CSPO aggregation. Parse / build CSPO model contexts / mappings / DOM. Factories streaming model context resources (chained CSPO streams) resolve new / existing contexts resources given input resources CSPO matching model resources streams CSPO augmented contexts (CUD).

APIs. XML:

XML / XSLT: XUL Application. Node: Augmentation Services Endpoint (I/O). Templates. Connectors. REST. DCI (XML code / data HAL I/O).

APIs. JavaScript:

JS / JSON / JSON-LD / Semantic / DOM / Functional QL (CRUD). TBD.

Input: RDF Quads.

ResourceRegistry: Mappings. Naming / Matching. Streams. Index. Functional Services.

Model Layers: StatementBuilder. Mappings.

Augment Models. Mappings. REST Context (referrer). Factory CSPO streams actions.

ResourceRegistry: Mappings. Naming / Matching. Streams. Index. Functional Services.

Model Layers: StatementBuilder. Mappings.

Output: RDF Quads (graph).

APIs: Facades. I/O Data CRUD Formats / Endpoints.

Inputs / Naming:

Input Quads: Statements from R2RQ, Apache Any23, Stanbol, MetaModel, Olingo, JBoss Teiid.

Resources Wrapper:

OntResource (Source URIs, Model URN);

Sources Alignment.

Contextual / Metadata URN:

CSPO URN IDs. DIDs.

Functional alignments (Mappings / Transforms / Layers).

Naming / Registry

IDs / Roles, Contexts:

Context Role Class (i.e: Dimension, Entity)

Context Role Class Instance

OntResource ID

CSPO ID

Metaclass

Class

Instance

Context

Occurrence

Role

urn:ont:[ID:ID][ID:ID][ID:ID]

URN IDs Syntax / Grammar.

Resolution: Traversal / Patterns / Predicates.

Blockchain: DHT CAN. Revisions: CUD / Versions.

Contracts: Mappings / Transforms.

Resources Monad<OntResource>.

IDs: Encoding (order / hierarchy). Aggregation. Dimensions. Bitstring lattices.

Patterns: Encoding Augmented with new knowledge still matching with previous one (Contexts, Subjects, etc.).

Augmented via Augmentation / functional data flow traversals aggregation (encoding).

Resource Monad

Statement Contexts

Wrapped Resource Type (Dimension, Unit, Measure)

Wrapped Resource Instance (Time, Minutes, 60)

Wrapped Resource Monad occurrence (Statement Context)

(TypeID, TypeInstanceID, WrappedResourceID, WrappedResourceOccurrenceID);

Statement resolves / matches IDs. Futures. Patterns, hierarchies, order resolution of Resources (data flow traversal build / match).

**Encoding**

To do. Achieve graph / graphs resources naming / ID scheme, one such allows for graph operations over an ID state space and operations over those IDs.

IDs. Hierarchy / Order: CSPO Selector patterns / IDs Encoding (Octal Quads Addressing):

Features encoding: Bitstrings / VSM / Layers / Hierarchies / Order operations: resolve metaclass, class, instance, context, occurrence, role.

Layers CSPO streams aggregation / merge. Context / roles pipelines.

CAN W3C DIDs. (Dimensional Addressing).

Inferences (Dimensional / Facts Models Augmentation):

Dialog. State Flows. Functional signatures / resources exchange activation.

PCN Aggregation Schema Model

PCN: Nested until primitives. Reification. Schema.

P: Previous

C: Context

N: Next

PCN: (PreviousLiteral : PCN, CurrentLiteral : PCN, NextLiteral : PCN);

Roles:

Models (Graphs schema and instance reify-able data) shows the following roles or characteristics according position (PCN) and dimension (referrer evaluating entity role:instance):

Context, Metaclass, Class, Instance, Occurrence, Role, Dimension.

As an example, using a quad based encoding, one could represent with an octal digit the base position / dimensional basic IDs features:

Previous: 100 (prev : S) : Occurrence / Domain / Kind / Meta class.

Prev / Curr: 110 (parent : OK) : Range signature.

Current : 010 (this : P) : Context.

Curr / Next: 011 (children : SK : SubjectType).

Next: 001 (next : O) : Range / Role.

Always: 111 (C: dimensions : Contexts) Meta class.

Complement: 101 : PK (ctx / roles XOR mask). Functional operations. CSPO masks / patterns (signatures). Context.

Masks: From Context, Metaclass, Type, etc. apply encoded mask: traversal (transforms / mappings).

Dataflow: Context mask (PKs O P, N) resolves Predicates matching Contexts S, O Types / Kinds. Functional Transforms, Mappings traversal / pipelines.

Populate network (primitives):

(TypeID, TypeInstanceID, WrappedResourceID, WrappedResourceOccurrenceID);

Populate network (sets models hierarchies):

(Type, Kind, Statement, Context);

P: Layers Statement Contexts.

C: Layers Statement Predicates.

N: Layer Statement Subjects / Object (Passion1 / Passion2 wipe materialize / parse statements).

Order:

Verb: Action, Passion, State.

State(A, P) :Transforms, Mappings.

(Context, Passion1, Action, Passion2);

Predicates / Predicate Types / Kinds:

(John : lover, loves, Mary : loved):

For a loved to exist there should exist first a lover. Dataflow: domain / range. PCN aggregation / recursion: axis / dimensions.

* Resource Monad.
* Transforms: to CSPO Role Types / from CSPO Role Types streams of resulting Contexts. Filters: predicates. Augmentation: aggregate / merge results streams. Contexts Mappings.
* Nested Monads wrapping contexts (map / flatten).
* Relation / Relationship: Reified statements / property graphs (encoding). PCN Resources Type / Value type / occurrences IDs.
* PCN: Relationship Models / SPO Relations I/O: (aggregate)
* C: Context. (Type)
* P/N Type: Subject. (Kind)
* PCN: Statement. (Statement)
* P/N Value: Object. (Context)
* Dimensional Values / Contexts / Types:
* (80, cm, distanceBetween)
* (80, years, ageUntilDate)
* Link Grammar.
* Types / Values: TMRM.
* 8m : Pop.
* Pop. : 8m.
* ex:Pat ex:age \_:x .
* > \_:x xsd:integer “75” .
* Reified SPOs.
* PCN: (PCN, PCN, PCN) : Resolves into keys.
* PCN: TypeID:ValueID (keys)
* Aggregates data. S/O. Kinds.
* PCN: KindID:ContextID.
* Aggregates schema. P. Kinds.
* PCN: VerbID:StateID
* Aggregates behavior. CK/PK.
* (Lover:Peter, ToLove:Loves, Loved:Mary) : Love:aLove.
* Predicate: property graph assertions (verb type / value). Verb type: infinitive. Declare domain / range / values (relationship attributes) shape. Value: verb instance. Reified Love.
* Predicates: S/O Functions / Mappings. Parsing: link grammar possible network statements (shapes / values fulfilment) assertions CUD. Retrieval: browse network links with navigation levels contexts / facets mappings.
* Resource Monad:
* Transforms: to CSPO Role Types / from CSPO Role Types streams of resulting Contexts. Filters: predicates. Augmentation: aggregate / merge results streams. Contexts Mappings.
* Previous: 100 (prev : S) : Occurrence / Domain / Kind / Meta class.
* Prev / Curr: 110 (parent : OK) : Range signature.
* Current : 010 (this : P) : Context.
* Curr / Next: 011 (children : SK : SubjectType).
* Next: 001 (next : O) : Range / Role.
* Always: 111 (C: dimensions : Contexts) Meta class.
* Complement: 101 : PK (ctx / roles XOR mask). Functional operations. CSPO masks / patterns (signatures). Context.
* Masks: From Context, Metaclass, Type, etc. apply encoded mask: traversal (transforms / mappings).
* Dataflow: Domain / Range.
* Populate network (primitives): (TypeID, TypeInstanceID, WrappedResourceID, WrappedResourceOccurrenceID);
* Populate network (sets models hierarchies):
* (Type, Kind, Statement, Context); CSPO Role. TBD.
* ((Peter knows Mary), (Peter loves Mary), (Peter MarriedWith Mary));
* Resource Monad.
* Statement Contexts.
* Wrapped Resource Type (Dimension, Unit, Measure).
* Wrapped Resource Instance (Time, Minutes, 60).
* Wrapped Resource Monad occurrence (Statement Context).
* No-SQL / LinQ
* Model:
* (Type, Kind, Statement, Context : CSPO Subject);
* Co-Algebra: Measures State / Flow / Behavior:
* (Dimension, Measure, Unit, Value);
* (Relationship, Relation, Kind, Entity);
* Monads, Zippers

Relation / Relationship: Reified statements / property graphs (encoding). PCN Resources Type / Value type / occurrences IDs.

PCN: Relationship Models / SPO Relations I/O: (aggregate: assertions augments Contexts).

PCN SPO to / from CSPO Models:

Context (Type, Dimension, Relationship) : Current type. (ToLove). Type: Layer Context role type(s).

Subject (Kind, Measure, Relation) : P/N Type. (Lover, Loved). Context Occurrence.

Predicate (Statement, Unit, Kind) : Current value type. (loves).

Object (Context, Value, Entity) : P/N Values. (Peter / Mary).

Tabular (I/O data flows streams):

(Type : CurrentType, Instance : CurrentInstance, Columns : P/N Types, Value : P/N Values);

(Context, Object, Attribute, Value);

P/N Types / Values : layered stack of wrappers. Functors mappings / transforms CUD / Retrieval reactive event model.

Populate PCN Context(s) x Layer(s) Dimensions.

Functional API Facade rendering front end shapes and flows (DCI / MVC Services / UI) data, schema and behavior ontology (upper system services / client facades interfaces).

Tabular I/O: (Type : CurrentType, Instance : CurrentInstance, Columns : P/N Types, Value : P/N Values);

PCN SPO to / from CSPO Models:

Relation / Relationship: Reified statements / property graphs (encoding). PCN Resources Type / Value type / occurrences IDs.

Relation:

Peter :marriedWith Mary;

Peter :marriedSince :date;

(etc…)

Relationship:

Peter :marriage :aMarriage

:aMarriage :husband :Peter

:aMarriage :marriedSince :date

Relationship:

Context (Type, Dimension, Relationship) : Current type. (ToLove).

Subject (Kind, Measure, Relation) : P/N Types Instances. (Lover, Loved).

Predicate (Statement, Unit, Kind) : Current value. (loves / loved). Properties (modalities, etc.).

Object (Context, Value, Entity) : P/N Values. (Peter / Mary).

IDs. Hierarchy / Order: CSPO Selector patterns / IDs Encoding (Octal Quads Addressing):

Features encoding: Bitstrings / VSM / Layers / Hierarchies / Order operations: resolve metaclass, class, instance, context, occurrence, role.

Layers CSPO streams aggregation / merge. Context / roles.pipelines.

CAN W3C DIDs. (Dimensional Addressing).

Inferences (Dimensional / Facts Models Augmentation):

Dialog. State Flows. Functional signatures / resources exchange activation.

PCN Aggregation Schema Model.

PCN: Nested until primitives. Reification. Schema.

* C: Context.
* P: Previous.
* N: Next.
* PCN: (Previous : PCN, Current : PCN, Next : PCN);
* Previous: 100 (prev : S) : Occurrence / Domain / Kind / Meta class.
* Prev / Curr: 110 (parent : OK) : Range signature.
* Current : 010 (this : P) : Context.
* Curr / Next: 011 (children : SK : SubjectType).
* Next: 001 (next : O) : Range / Role.
* Always: 111 (C: dimensions : Contexts) Meta class.

Complement: 101 : PK (ctx / roles XOR mask). Functional operations. CSPO masks / patterns (signatures). Context.

Masks: From Context, Metaclass, Type, etc. apply encoded mask: traversal (transforms / mappings).

Dataflow: Context mask (PKs O P, N) resolves Predicates matching Contexts S, O Types / Kinds. Functional Transforms, Mappings traversal / pipelines.

Populate network (primitives):

(TypeID, TypeInstanceID, WrappedResourceID, WrappedResourceOccurrenceID);

Populate network (sets models hierarchies):

(Type, Kind, Statement, Context);

C: Layers Statement Contexts.

P: Layers Statement Subjects. Previous Contexts.

N: Layer Statement Objects. Next Contexts.

Order:

Verb: action, passion, state. S(A, P). Transforms, Mappings signatures.

Predicates / Predicate Types / Kinds:

(john : lover, loves, mary : loved):

For a loved to exist there should exist first a lover. Dataflow: domain / range. PCN aggregation / recursion: axis / dimensions.

* Resource Monad.
* Transforms: to CSPO Role Types / from CSPO Role Types streams of resulting Contexts. Filters: predicates. Augmentation: aggregate / merge results streams. Contexts Mappings.
* Nested Monads wrapping contexts (map / flatten).
* Relation / Relationship: Reified statements / property graphs (encoding). PCN Resources Type / Value type / occurrences IDs.
* PCN: Relationship Models / SPO Relations I/O: (aggregate)
* C: Context. (Type)
* P/N Type: Subject. (Kind)
* PCN: Statement. (Statement)
* P/N Value: Object. (Context)
* Link Grammar.
* Types / Values: TMRM.
* 8m : Pop.
* Pop. : 8m.
* ex:Pat ex:age \_:x .
* > \_:x xsd:integer “75” .
* Reified SPOs.
* PCN: (PCN, PCN, PCN) : Resolves into keys.
* PCN: TypeID:ValueID (keys)
* Aggregates data. S/O. Kinds.
* PCN: KindID:ContextID.
* Aggregates schema. P. Kinds.
* PCN: VerbID:StateID
* Aggregates behavior. CK/PK.
* (Lover:Peter, ToLove:Loves, Loved:Mary) : Love:aLove.
* Predicate: property graph assertions (verb type / value). Verb type: infinitive. Declare domain / range / values (relationship attributes) shape. Value: verb instance. Reified Love.
* Predicates: S/O Functions / Mappings. Parsing: link grammar possible network statements (shapes / values fulfilment) assertions CUD. Retrieval: browse network links with navigation levels contexts / facets mappings.
* Resource Monad:
* Transforms: to CSPO Role Types / from CSPO Role Types streams of resulting Contexts. Filters: predicates. Augmentation: aggregate / merge results streams. Contexts Mappings.
* Previous: 100 (prev : S) : Occurrence / Domain / Kind / Meta class.
* Prev / Curr: 110 (parent : OK) : Range signature.
* Current : 010 (this : P) : Context.
* Curr / Next: 011 (children : SK : SubjectType).
* Next: 001 (next : O) : Range / Role.
* Always: 111 (C: dimensions : Contexts) Meta class.
* Complement: 101 : PK (ctx / roles XOR mask). Functional operations. CSPO masks / patterns (signatures). Context.
* Masks: From Context, Metaclass, Type, etc. apply encoded mask: traversal (transforms / mappings).
* Dataflow: Domain / Range.
* Populate network (primitives): (TypeID, TypeInstanceID, WrappedResourceID, WrappedResourceOccurrenceID);
* Populate network (sets models hierarchies):
* (Type, Kind, Statement, Context); CSPO Role. TBD.
* ((Peter knows Mary), (Peter loves Mary), (Peter MarriedWith Mary));
* Resource Monad.
* Statement Contexts.
* Wrapped Resource Type (Dimension, Unit, Measure).
* Wrapped Resource Instance (Time, Minutes, 60).
* Wrapped Resource Monad occurrence (Statement Context).
* No-SQL / LinQ
* Model:
* (Type, Kind, Statement, Context : CSPO Subject);
* Co-Algebra: Measures State / Flow / Behavior:
* (Dimension, Measure, Unit, Value);
* (Relationship, Relation, Kind, Entity);
* Monads, Zippers

Cons lists:

Hierarchycal encoded ordered tree. Hashing. CAM. Paths (dimensions / axis / property graphs.

(Person:Husband (Married:Marriage (Person:Wife);

SPO Pairs Scrolls Layers / Link Grammars hierarchies. Context, metaclass, class, instance, occurrence, roles scrolling in / out according current position traversal (shapes / paths / zipper patterns).

Encode Graph as hierarchical aggregated Lists:

Aggregate Context Relationships types (Marriage Statements), Relation types instances (aMarriage Marriage Statements), Kind roles, Entity players.

Relation / Relationship: Reified statements / property graphs (encoding). PCN Resources Type / Value type / occurrences IDs.

PCN: Relationship Models / SPO Relations I/O: (aggregate: assertions augments Contexts).

PCN SPO to / from CSPO Models:

Context (Type, Dimension, Relationship) : Current type. (ToLove). Type: Layer Context role type(s).

Subject (Kind, Measure, Relation) : P/N Type. (Lover, Loved). Context Occurrence.

Predicate (Statement, Unit, Kind) : Current value type. (loves).

Object (Context, Value, Entity) : P/N Values. (Peter / Mary).

Tabular (I/O data flows streams):

(Type : CurrentType, Instance : CurrentInstance, Columns : P/N Types, Value : P/N Values);

(Context, Object, Attribute, Value);

P/N Types / Values : layered stack of wrappers. Functors mappings / transforms CUD / Retrieval reactive event model.

Populate PCN Context(s) x Layer(s) Dimensions.

Functional API Facade rendering front end shapes and flows (DCI / MVC Services / UI) data, schema and behavior ontology (upper system services / client facades interfaces).

Tabular I/O: (Type : CurrentType, Instance : CurrentInstance, Columns : P/N Types, Value : P/N Values);

PCN SPO to / from CSPO Models:

Relation / Relationship: Reified statements / property graphs (encoding). PCN Resources Type / Value type / occurrences IDs.

Relation:

Peter :marriedWith Mary;

Peter :marriedSince :date (etc…)

Relationship:

Peter :marriage :aMarriage

:aMarriage :husband :Peter

:aMarriage :marriedSince :date

Context (Type, Dimension, Relationship) : Current type. (ToLove).

Subject (Kind, Measure, Relation) : P/N Types Instances. (Lover, Loved).

Predicate (Statement, Unit, Kind) : Current value. (loves / loved). Properties (modalities, etc.).

Object (Context, Value, Entity) : P/N Values. (Peter / Mary).,

Cons lists:

Hierarchycal encoded ordered tree. Hashing. CAM. Paths (dimensions / axis / property graphs.

(Person:Husband (Married:Marriage (Person:Wife);

SPO Pairs Scrolls Layers / Link Grammars hierarchies. Context, metaclass, class, instance, occurrence, roles scrolling in / out according current position traversal (shapes / paths / zipper patterns).

Encode Graph as hierarchical aggregated Lists:

Aggregate Context Relationships types (Marriage Statements), Relation types instances (aMarriage Marriage Statements), Kind roles, Entity players.

Autoencoder: IO Ordered Streams CAM. Prev / Curr / Next List Model State. Classification Requests Order relevant to Predictions. Ontology Matching: encode dimensional / relations: Model.

Augmentation: List Model prev / next Activation (Kind Roles), Aggregation (Model Layers), Alignment (Properties / Relations Attributes).

**Data flow:**

Mappings: Create, Update, Delete (CUD).

(Mapping, DomainSelector : Kind, Input : URI, RangeSelector : Kind);

Transforms: Retrieve.

(DomainSelector : Kind, Input : URI, RangeSelector : Kind, Value : URI);

Functional Mapping: Value specification.

Functional Transform: Mapping Value.

Mappings / Transforms are declaratively stated in Meta Model meta resources (context / selector, object / selector, transform, values). Values updated in model references (slots).

Mappings / Transforms declares shapes / instances (type / value slots) of reified types / values. Kinds: Data flow Value slots. Shapes, Templates.

Input / Output:

(Context, Object, Attribute, Value);

tabular / meta model I/O / alignments translation interfaces. Handle CRUD semantics (messages). Functional REST / HATEOAS (DCI core transforms / mappings) message based model browsing. Mapping Values: data flow slots.

Layers Matrices fulfills Mappings / Transforms. Perform Augmentations: Inference, matching, alignment, learning populating Mappings / Transforms. Example: Kind Context layer aggregates Kinds Transforms / Mappings from matrix layers occurrences. Kind as result of a Subject Transform.

Mapping / Transforms Kinds types:

CSPO Dimensional layers.

Context

Subject

Predicate

Object

ContextKind

SubjectKind

PredicateKind

ObjectKind

Kind Mappings:

(ContextKind, Occurrence, Attribute : Kind, Value);

Attribute: Predicate / super Kind;

(SubjectKind, Occurrence, Attribute : Kind, Value);

Attribute: Predicate / super Kind;

(PredicateKind, Occurrence, Attribute : Kind, Value);

Attribute: Predicate / super Kind;

(ObjectKind, Occurrence, Attribute : Kind, Value);

Attribute: Predicate / super Kind;

Augmentations:

Mappings signatures: (DomainKind, RangeKind). Data flow streams. Reactive event driven (Resource functor core transforms).

Aggregations:

Subjects Streams. Context aggregated Subject Occurrences stream. Mappings driven transform.

Mapping: (ContextKind, SubjectKind);

Activations:

Predicate Streams. Subject aggregated Predicate Occurrences stream. Mappings driven transform.

Mapping: (SubjectKind, PredicateKind);

Alignments:

Object Streams. Predicate aggregated Object Occurrences stream. Mappings driven transform.

Mapping: (PredicateKind, ObjectKind);

Functional Augmentation streams: Augmentation streams concatenation (functors pipelines). Apply domain / range signature matching Transforms. Collect / merge of CSPO aggregated Mappings contextual Transforms augmented statements. Populate matrices. Populate Mappings / Transforms.

Transforms / Mappings:

Augmentations

Metaclass

Class

Instance

Occurrence

Context

Role

Others

Kinds: Domain / Range Types

Schema Kinds: Meta Model / CSPO

StatementKind

OccurrenceKind

KindKind

URIKind

Semiotic Kinds: Input / Output

ContextKind

ObjectKind

AttributeKind

ValueKind

Domain Kinds:

RelationshipKind

RelationKind

KindKind

EntityKind

Dimensional Kinds:

DimensionKind

MeasureKind

UnitKind

ValueKind

Schema Kinds: Reified Kind Values.

Domain Kinds: Kind types instances.

Transform Invocation contexts:

Kind: Aggregated Kind occurrences.

Kind / Value: Value aggregated Kind occurrences.

Value : Aggregated Value occurrences.

EntityKind instance in Subject, Predicate, Object role Mapping occurrences. Kind Value Attributes / Values according domain data flow context state (Transform Kind Value).

Example:

(Marriage, Wedding, Invited : Kind, Entity);

Kind 'Invited' as a statement Context role (Invited Context Layer) maps / transforms to all 'Invited' entities. Mappings multiple values.

Kind 'Invited' as a Marriage Wedding statement Predicate maps / transforms to given Marriage Wedding instance 'Invited' entities. Mappings multiple values.

**Augmentation**

Aggregation: Keys. Identify Key / Equivalence (in some Domain axis) Predicates. Domain / Range schema. Object functional values. Alignment. Activation. Subjects Key Predicate to same Object: Subjects equivalence.

i.e., if the same resource has multiple “first” arcs from it then those nodes must be equivalent.

Functional (wrappers) List encoding of Layer Quads. Order: Types: sub / super inheritance, Keys: Aggregation / Properties hierarchies. Grammar: context / context addressable facts.

Kinds (Attributes / Values):

Subject Kind: (Context, Subject, Attribute, Value);

Predicate Kind: (Context, Attribute, Predicate, Value);

Object Kind: (Context, Attribute, Value, Object);

Type hierarchy inference due aggregation of multiple occurrences of the same attributes together.Kinds:

S: ?; SK: class (employee);

P: ?; PK: context (worksFor);

O: ?; OK: role (employer);

Ordered hierarchies: Kinds properties sets inclusion relationship. Sequential ordered sets type / subtype relationship. Sub type "next" in order relationship. Context domain / range assertions.

Alignment / Dataflow: ordered domain / schema assertions: Contexts metadata aligns matching Subjects / Objects resolving Context Statements.

Services: GraphQL. Property Graphs. Distributed Naming (blank / new nodes uniform hashing), Index and Registry Services.

Service Predicate URIs: P(S) : O. Augmentation expansions. Value types / reference types.

Naming Os: All Subject names occurrences in Statements.

Registry Os: All Subject occurrences as Predicate Object (role) values.

Index Os: All Subject Predicate hierarchies graph Statements.

Dimensional roles: N as :age, N as :distance. Relation / Relationship Interaction Domain Flows model layers.

**Protocol:**

HTTP REST HATEOAS RDF I/O. State flows: transaction workflow data encoded in request / response interaction state (Gestures: application / domain upper) messages. Augmentation.

Functional browsing. Facets, variables, wildcards, placeholders. Prompts. Runat code / data resolution semantics. Content addressable (DHT / CAN) dataflow browsing.

Rendering: declarative clients transaction state encoding specifications. Templates. Layers: Model, Service, Client translation / transforms (Templates / Gestures: upper / application / domain Model Entities). Messages DOM: DCI Interaction Graph. Annotated Addressable Interactions (objectives / rels: context / roles / goal / purposes flow items tagging).

**Framework:**

CAN: Content Addressable Network (Distributed Hash Table):

Zones: Nodes assigned Virtual n-dimensional Coordinate Space portions.

2-Dimensional spaces: Quads matrix zones (CSPO x OPSC axis).

4-Dimensional spaces: CSPO axis zones.

Neighbors: Node Join (Axis partitioning) Similarity / routing point P, Split P Zone (axis), assign Zone to Node.

Point: Node Zone Put (key, value). Similarity / routing point P dimensional distance.

Key: Content dimensional points. Similarity / hashing. Adjacency (neighbours / routing distance) by CSPO Quads matrix sides (torus).

To do: DHT Blockchain / W3C DIDs (Distributed IDs).

**Domains Modelling:**

Upper abstract domain modeling classes. Abstract use cases concepts applicable in different domains integration such as e-commerce, health-care or any other business organizational structures domains.

Semantic Dimensional Encoding (align abstract upper ontology levels, i.e.: Interactions / Gestures). StratML interoperability.

Alignment to / from Dimension, Measure, Unit, Value / Meta Model Layers. DCI Flows / Data flow (order / reactive).

Classes:

* Needs
* Good
* Product
* Item
* InventoryItem
* Supplier
* Producer
* Consumer
* Ownership
* Transaction
* Context (Campaign)
* Goals / Purposes
* Exchange (Transaction specification)
* Protocol: State Representations exchanges (Contexts Roles Facets / Levels)

**Models: Sets, Individuals, Mappings**

Models which are instances of the Sets Layers Model Structure. Model Properties:

Metaclass, Class, Instance, Occurrence, Context, Role, Attribute, Value.

Functional. Mappings / Transforms. T-Box / A-Box. Sets, Groups, Categories: TBD.

Types Model:

Types (types in sets roles):

(Relation : Statement, Relationship : Kind, Role : SPO, Dimension : U);

(Dimension, Context, Class, Resource);

(Resource, Dimension, Context, Class);

(Class, Resource, Dimension, Context);

(Context : Statement, Class : Kind, Resource : SPO, Dimension : U);

Individuals Model:

Individuals (individuals / sets types instances):

(Context : Statement, Class : Kind, Resource : SPO, Dimension : U);

(Dimension, Context, Class, Resource);

(Resource, Dimension, Context, Class);

(Class, Resource, Dimension, Context);

(Context : Statement, Class : Kind, Resource : SPO, Dimension : U);

Mappings Model:

Mappings (type / individual relationships):

(Context : Dimension, Occurrence : Measure / SPO, Attribute : Unit / Kind, Value : Value / Statement);

Models metadata, properties and upper alignments / augmentations relationships Model data.

(Value, Context, Occurrence, Attribute);

(Attribute, Value, Context, Occurrence);

(Occurrence, Attribute, Value, Context);

(Context, Occurrence, Attribute, Value);

**Layers: Augmentations / Inference**

Activation: Classification (Context types Occurrences Attributes).

Activation: Which Attributes has Context Occurrence (according to its Kind in Context / Role) in this Occurrence.

Alignment: Regression (Context types Occurrences Attributes Values).

Alignment: Context Occurrence Attributes Values (according to its Kind in Context / Role).

Aggregation: Clustering (Context types Occurrences).

Aggregation: Context type instance aggregates type instance child Occurrences (parent Context type instances) matching grouping criteria (Encoding).

Augmentations:

(Context, Occurrence) : Value;

Activation:

(Statement, Resource) : Kind;

Alignment:

(Kind, Statement) : Resource;

Aggregation:

(Resource, Kind) : Statement;

**Model Semantics:**

Data: Individuals. Mappings. Data Occurrences Aggregation.

Data: Individuals Model.

(Dimension, Context, Class, Resource);

Mappings (type / individual relationships):

(Context : Dimension, Occurrence : Measure / SPO, Attribute : Unit / Kind, Value : Value / Statement);

Information: Types. Mappings. Type Occurrences Attributes. Activation.

Information: Types Model. Schema.

(Dimension, Relation, Relationship, Role);

Mappings (type / individual relationships):

(Context : Dimension, Occurrence : Measure / SPO, Attribute : Unit / Kind, Value : Value / Statement);

Knowledge: Individuals / Types Mappings (Attributes) Values. Alignment.

Knowledge: Behaviors.

Mappings (type / individual relationships):

(Context : Dimension, Occurrence : Measure / SPO, Attribute : Unit / Kind, Value : Value / Statement);

**Ontology Matching: Relations / Relationships**

Entity Relationship instance asserted as a reified concept with its type and attributes or as a series of triple statements which describes the given Entity Relationship instance via individual assertions. Bidirectional translation.

aPerson loves anotherPerson.

Person loverOf Person.

loverOf predicate: Kind of aPerson. Domain / Range. Dataflow (Functional Augmentations).

Loving: loverOf Kind.

aLoving: loves Kind.

TBD: Relationship / Relation

Reify Kinds as SPOs : Types Model

Reify Statements as / Kinds / SPOs : Mappings Model

Augmentations (Aggregation).

NLP NER.

CEP:

Cause / Effect.

Periodical (units lengths).

Order inference.

Hierarchies. Containment (Graph).

Dimensions (axis / measures).

Facts. Relationships (events, assertions).

Translation: Distance, speed, time. Dimension / axis products: relationships.

Autoencoder: IO Ordered Streams CAM. Prev / Curr / Next List Model State. Classification Requests Order relevant to Predictions. Ontology Matching: encode dimensional / relations: Model.

Augmentation: List Model prev / next Activation (Kind Roles), Aggregation (Model Layers), Alignment (Properties / Relations Attributes).

**Ontology Matching: Dimensional Alignments (Mappings):**

Explain Layer Context, Occurrence, Attribute, Value Pattern for Models SPO Statements functional mappings expansion:

(Context, Occurrence, Attribute, Value);

For a given CSPO Quad:

(C, S, P, O);

Expansion:

(C, P, S, O);

(C, O, P, S);

TBD.

Mappings (set / individual relationships):

(Context : Dimension, Occurrence : Measure, Attribute : Unit, Value : Value);

Order. Comparison. Relations. Upper Ontology assertions. Augmentations. TBD.

Relation / Relationship: Tabular / OGM (Object Graph Mapper):

I/O: (Class, ClassID, Attribute, Value);

Class: Table / Object Type.

ClassID: PK / Object ID.

Attribute: Column / Member.

Value: Cell / Field Value.

Subject Kind: Relation / Domain.

Predicate Kind: Relationship.

Object Kind: Mapping / Range.

Dataflow: Reactive Functional Augmentation / Integration APIs.

Indices: Apply functional mappings expansion.

NLP NER.

CEP:

Cause / Effect.

Periodical (units lengths).

Order inference.

Hierarchies. Containment (Graph).

Dimensions (axis / measures).

Facts. Relationships (events, assertions).

Translation: Distance, speed, time. Dimension / axis products: relationships.

Autoencoder: IO Ordered Streams CAM. Prev / Curr / Next List Model State. Classification Requests Order relevant to Predictions. Ontology Matching: encode dimensional / relations: Model.

Augmentation: List Model prev / next Activation (Kind Roles), Aggregation (Model Layers), Alignment (Properties / Relations Attributes).

**Functional API: Monads / Transforms**

Resource / Layer?

Context / Occurrence / Mapping?

Mapping: Selector Monad. Matching Resource / Role set?

Context / Occurrence Monads wrapping Layers Hierarchy Contexts.

Entity Alignment / Matching resolution via Functional Augmentations: Agggregations / Activation / Alignments (upper / dimensional matchings). Versioned graph: stateless / functional. Mappings assertions matching.

APIs: Augmentations, Query, Traversal, Matching, Transforms. Functional APIs Query / Browse / Traversal / Transforms examples. Encoding / Matching.

**Encoding: Functional Mappings**

Masks: Predicates of Set memberships. Functional Mappings. ID encoded state / transforms. Models merge. Ontology Matching. Mappings Model: Types / Instances Models merge (upper) Augmentations.

Mappings / Functional Encoding: Upper Dimensional Matchings / Augmentations. Mappings Model masks matchings reflects / leads to Types / Individuals Models Augmentations / Assertions.

Mappings / Functional Encoding: Relation Statements / Relationship views / matchings examples.

Mappings Upper Alignments examples (dates, marital status, hiring). Relation Relationship statements order / context properties (Dimensional Alignments).

IDs: metaclass, class, instance, context, role, occurrence, previous, next ID roles relations for Model Set Contexts.

Augmentations / Transforms: Model / Domains functional mappings. Order. Dimensions. Axes. Flows. Hierarchies. Inference / Population.

Levels: Augmentations. Mappings.

Levels: Resource, Kind, Statement.

Levels: Reify Statement as Kind, Kind as Resource, Resource as Statement.

Levels: Reify Resource as Kind, Kind as Statement, Statement as Resource.

**TODO Items:**

* Dimensions Encoding: Given Dimensional Contexts (CSPO Models set layouts) having four dimensional sets (Types Model, Individuals Model, Mappings Model, State Model) each representing (nested) CSPO inputs / parts of a recursively aggregated CSPO layout (i.e. aggregated layout Context is Mappings Model, Subject is State Model, etc.) having this setting (Models types / layers class / instance IDs) reified in this fifth "Focus" Model which represents a "snapshot" of current state and available transitions (Focus shifts).
* Models: CSPO Layers (matrix) layout.
* Focus Mapping Model. Axes (X / Y: Model instances matrices, cycles), intersection (Z: Model instance matrix):
* Model patterns:
* (Dimension, Unit, Measure, Value);
* (Context, Occurrence, Attribute, Value);
* Context / Dimension / Context:
* Occurrence / Unit / Subject:
* Attribute / Measure / Predicate:
* Value / Value / Object:
* X Model: Context / Schema / Information / Relationships
* Y Model: Data / Relations
* Z Model: Interaction / Context instance Data state calculated intersection. Behavior
* Upper Y / Lower Y: Previous / next data state.
* Left X / Right X: Previous / next context state.
* Augmentations calculate current, previous, next Model states.
* ImplementationItems & drafts documents topics.
* Diagrams (TO DO):

**Contents**

Objectives: Develop Protocol (APIs) to facilitate Enterprise Application Integration (EAI) by means of Semantic technologies and Machine Learning. Ontology matching driven data, schema, behavior inference / aggregation / matching. Reasoning and learning over different consolidated backends alignments.

Distributed P2P (Blockchain) approach of data synchronization between peers for ease of deployment patterns election and datasources integration (APIs, microservices, etc.).  
  
Data alignment:  
  
Determine if two instances (example: records) of two different backends or services refer to the same entity (Customers : John D. / Employees : John Doe).  
  
Schema alignment:  
  
Determine, for example, meaning and equivalences between diverse (aggregated / composite) schemas (equivalent classes, equivalent attributes, equivalent roles).  
  
Behavior alignment:  
  
Determine meaning and equivalences between (aggregated / composite) behavior contexts and behavior contexts invocations / interactions (Appointment / Interview, anAppointment / anInterview. Behavior flows aggregated from backends / services learning).  
  
Layered models semantic infrastructure for integration of heterogeneous backends (meta models).

Alignments Augmentations:  
  
Activation: type inference : classification (determine class / metaclass / roles for entity attributes and values).  
  
Activation infer attributes / relations : clustering (from multiple occurrences of same entity in diverse data sources).  
  
Aggregation: infer roles in contexts: regression (Person class in Employment interaction : Developer role).  
  
Integration of addressable resources. Reactive I/O (sync back ends). Content type driven semantic augmentation / annotations.  
  
Integrated view. Navigate contexts, data, interactions. APIs. Dimensional views annotations (analysis / mining).  
  
Augmentation of distributed resources. Annotations (Semantic / ML). API for resource / schema / interactions exploration / protocol for message based API "dialogs" execution. HAL (Hypertext Application Language), OData (REST) like interfaces.  
  
Example: Google Drive / Google Knowledge Graph APIs Augmented with ML / Semantic intelligence tailored for specific domains / application kinds.

**Features**

Graph encoding of data / schema / behavior. Dimensional / Grammar annotations. MetaGraph: augmentation / transforms (Messages). Features.

Parallel distributed graphs models augmentation / transforms synchronization (Messages). Event sourcing (distributed inferences). P2P / DIDs.

Augmentation. Ontology matching. Hypermedia augmentation protocol. Browser / Client APIs.

URIs API for annotating network retrievable resources metadata. Content type / model driven augmentations / activations (models features / outputs). Subject attributes / values. Occurrences contexts / roles. Paths, pointers, locators. Example: annotate document URIs (parts, sections, mentions), annotate images URI (whole image description, coords: classes, individuals), annotate DB, table, row, column, value URIs, annotate / describe service / APIs URIs. Hypermedia protocol composable with other (described / annotated) APIs / resources. Example: Drive APIs.

**RDF triples, quads introduction**

RDF Models: rdfs type, class, subClassOf, sameAs, reification when appropriate. RDFS. OWL (alignments).

RDF / OWL Backend: APIs. Details: Contents triples / models introductions.

Turtle. N3.

Example: feed Dimensional model for equivalences (units), comparison (orders).

TBD.

**Models: Quads, Contexts, Occurrences, Attributes, Values.**

Declarative means of using RDF quads to state application object models (data, schema and behavior).

Aggregation.  
Kinds.  
Grammar.

Formalization: Functional / Object API. Reference / Data model. Sets, categories, models.

Subjects: attributes / values, contexts / roles.

(Context, Occurrence, Attribute, Value);  
(Context, Sign, Concept, Object);

Instance, occurrence, class, metaclass.

Hierarchies: layered quad statements are represented by a class hierarchy which root is the Resource<T> monad. There is a subclass relationship between each layer implementing class and the one of the next layer (Dynamic Object Model).

Quads in the context role of lower layers represents occurrences of context enclosing layer. Assert class hierarchies, order relation (temporal, causal, containment, etc.) by attrs / vals, set / superset relations.  
Discovery: All model kinds are browseable / discoverable.

Determine class (reified layers contexts) hierarchies:  
(ContextReifiedClass, ContextReifiedSubClass, SubClassAttributeKind, SubClassValueKind);

Merge / specify model, context, interaction graphs. Reified model resources, statements, kinds. Model, context, interaction model graphs layers specifications. Reified models layers contexts resources describe graphs. Augmentation. Message context statement occurrence: Model.

Model MetaGraph: Resource, Statement, Kind (reifying class / instances) contexts / occurrences / attributes / values. Encoding. Message dispatch,  
event bus routes. URIs / IDs mappings.Resource set specification resolution. Resolve concrete resources, Message expansion. Resolve Message / dialog (CRUD) semantics.  
Dimensional / Grammar models.

TBD.

**URIs, Resource, Statement, Kind APIs**

TBD.

Message service URIs: contextual (statement / dialog) service invocations.

Example: Subject (image URI / resource : source), Predicate (detection service / index service), Object (detection / search results endpoint / placeholder : destination).

Grammars: Predicate Kind (face / search recognition signature) from Subject (faces images / names) / Object (face classes / subjects) Kinds. Kind model layers.

Models definition: data (Statement, Entity), schema / context (Role, Class), interactions / behavior (Flow, Behavior).

Kinds / Roles:  
Grammar: kinds layers aggregation (CSPO layers Kinds).  
Layers: Roles (Models metaclass context resources).

Reified Kind: (Kind, Occurrence, Attribute, Value);  
Grammar input set model specificatíon (Statement layer kinds).

Dimensional input set model specificatíon (from Statement layer, ordered SPOs: order criteria, comparisons. Kinds / class / occurrence / instance  
order criteria?). Value, Previous, Distance, Next. Dimension, Unit, Measure, Value (aggregated ordered statements).

MetaGraph (resolution). Dimensional / Grammar alignments / annotations.

Model MetaGraph: Resource, Statement, Kind (reifying class / instances) contexts / occurrences / attributes / values. Encoding. Message dispatch,  
event bus routes. URIs / IDs mappings. Resource set specification resolution. MetaGraph resolves concrete resources, Message expansion. Resolve Message / dialog (CRUD) semantics via MetaGraph driven transforms (data / schema / behavior augmentation: dialogs).

URIs API for annotating network retrievable resources metadata. Content type / model driven augmentations / activations (models features / outputs). Subject attributes / values. Occurrences contexts / roles. Paths, pointers, locators. Example: annotate document URIs (parts, sections, mentions), annotate images URI (whole image description, coords: classes, individuals), annotate DB, table, row, column, value URIs, annotate / describe service / APIs URIs. Hypermedia protocol composable with other (described / annotated) APIs / resources. Example: Drive APIs.

Resource<T : URI> monad. Message functors. Transform reactive extensions.

Transform : Observer / Observable of Resource<T : URI>. Stream. Built upon Resources / Messages (TransformBuilder).

Identity and other core transforms (core messages). Stream.

flatMap(Message::apply) : Transform<Resource<R : URI>>.

API: Class for layer for model.  
API: Class for layer (DOM).  
API: Parameterized Resource: layer classes determined by URIs hierarchy, i.e.: Resource<Entity>, Entity : URI.

Base core services URIs (index, naming, registry). URI subclasses implementing / wrapping state for Resource monads offering protocols / addressing / content types / representations facades for services: DBs, WS (REST, SOAP, SPARQL), ML (predictions), etc.

Discovery: All model kinds are browseable / discoverable.

Determine class (reified layers contexts) hierarchies:  
(ContextReifiedClass, ContextReifiedSubClass, SubClassAttributeKind, SubClassValueKind);

Merge / specify model, context, interaction graphs. Reified model resources, statements, kinds.

Model, context, interaction model graphs layers specifications. Reified models layers contexts resources describe graphs. Augmentation. Message context statement occurrence: Model.

MetaGraph (resolution). Dimensional alignments / annotations.

Model MetaGraph: Resource, Statement, Kind (reifying class / instances) contexts / occurrences / attributes / values. Encoding. Message dispatch, event bus routes. URIs / IDs mappings.Resource set specification resolution. Resolve concrete resources, Message expansion. Resolve Message / dialog (CRUD) semantics.

**Model Layers**

See Messages / Augmentation.

Composed of quads semantically aggregated into layers.

Core features provides:  
Alignment  
Activation  
Aggregation

Message / Transform driven specification of Alignment, Activation, Aggregation (Augmentation).

Message (Resource set expression);

Message: Model parent layer (Resource). Nested Messages CSPOs.

Model: Message scopes. Described as (nested) Message Resource set expressions.

Models: Data (Models), Contexts (Grammars), Interactions (MetaGraph : Models / Grammars bindings). Dimensional annotations.

MetaGraph: Resource, Statement, Kind class / instance as CSPO MetaGraph statement roles. Class / subclass relationship, Kind / subkind relationship.

Reactive nodes (Message events):

. Input Message event;

. Augmentation;

. Model / MetaGraph Message resolution (grammars / models / backends / services); Model Resource(s) response activation;

. Augmentation (Message : response / dialog);

. Output Message (events);

Grammars.

Upper / Dimensional ontology.

Inter models alignments.

Services (Endpoint URIs: Resource facades).

Reified model resources (CSPO, Resource, Statement, Kinds, Layers). Augmentation (Alignment, Activation, Aggregation) Messages / Transforms.

Model, Contexts, Interactions IO:  
Model: aggregated resource statements.

Context: aggregated model kinds (grammar statements).

Interaction: aggregated model / context bindings.

Inputs: resource statements, resolvable messages. Operation semantics (CRUD, browse, etc.) according input statements layout. Model endpoint. Materializes input resource statements and fully resolved message resource statements from interactions applying Augmentation and matching messages transforms.

Resource flow: input plain RDF URIs statements. Model / Context updates. Transform matches concrete resources.  
Resource flow: input message URIs statements. Context / Interaction perform. Transform matches resources in messages context grammar kinds hierarchies.

Outputs: resource statements with possible further resolvable messages (Model IO recursion / dialogs). Interaction queries context / model back for further resolutions. Message transform stream with request message applied plus matching context resolved resources from message.

Model, Contexts, Interactions IO:

Resource, Statement, Kind, Message, Transform.

Subscription, Subscriber, Producer, Consumer, Processor.

Model: aggregated resource statements model.

Context: aggregated model kinds (grammar statements model).

Interaction: aggregated model / context / dialogs bindings model.

Aggregation: layers. Parameterized Resource<C, S, P, O> : CSPO : URIs hierarchy.

Materialized interactions re-populate model and context (Augmentation). Browse context model: kinds and grammar known statement "templates" (by kinds hierarchy layers aggregation) navigation for discovery of domain messages resource kinds.

Model, Context, Interaction IO: Message. Nested CSPO contexts quad, CSPO resources (plain URIs, kinds, nested contexts). Wildcards, variables, placeholder, null values: Message structure defines CRUD behavior.

Message: Resource model hierarchy parent class (monad of plain URI, parameterized resources). Resource set specification. Any Resource is a Message, specifying a potential set of other Message (Resource) in a model (layers).

Resource : Message. Resource resolution: known URIs, known resource kinds bindings, dialog (resource set specification) recursively. Interaction model (dialog resolved resources set). Wildcards, variables, placeholder, null values: Message structure defines CRUD behavior.

Resource monad of URIs or Message monad of Resource?

**Data Model**

Data Model layers population / augmentation.

(Resource, Resource, Resource, Resource) : Resource / Message (Model).  
(Entity, Subject, Attribute, Value);  
(Role, Entity, Attribute, Value);  
(Kind, Role, Entity, Attribute);  
(Class, Kind, Role, Entity);  
(Flow, Class, Role, Entity);  
(Behavior, Flow, Class, Role);

Messages (Model : Resource) as Resource set specifications. Subject, Attribute, Value : Resource.

Determine class (reified layers contexts) hierarchies:  
(ContextReifiedClass, ContextReifiedSubClass, SubClassAttributeKind, SubClassValueKind);

Merge / specify model, context, interaction graphs. Reified model, resources, statements, kinds.  
Model, context, interaction model graphs layers specifications. Reified models layers contexts resources describe graphs. Augmentation. Message context statement occurrence: Model.

Model MetaGraph: Resource, Statement, Kind (reifying class / instances) contexts / occurrences / attributes / values. Encoding. Message dispatch,  
event bus routes. URIs / IDs mappings. Resource set specification resolution. MetaGraph resolves concrete resources, Message expansion. Resolve Message / dialog (CRUD) semantics via MetaGraph driven transforms (data / schema / behavior augmentation: dialogs).

Models definition: data (Statement, Entity), schema / contex (Role, Class), interactions / behavior (Flow, Behavior).

Kinds / Roles:  
Grammar: kinds layers aggregation (CSPO layers Kinds).  
Layers: Roles (Models metaclass context resources).  
Reified Kind: (Kind, Occurrence, Attribute, Value);

**Schema Model (Grammars)**

Schema Model layers population / augmentation.

Grammar Resource input set model specificatíon (Statement layer kinds Messages).

Grammars: Predicate Kind from Subject / Object Kind. Kind model layers.

**Behavior Model (Dimensional annotations)**

Dimensional Model layers population / augmentation. Purpose modelling. Dimensional Concepts.

Order layers statements. Hierarchies (contexts / kinds). Parent / child relationships (steps). Order type relationships: husband: single / marriage / married.

(Value, Previous, Distance, Next); Person, Single, Marriage, Married; Man, Single, Marriage, Husband; Woman, Single, Marriage, Wife.  
(Measure, Value, Previous, Distance);  
(Unit, Measure, Value, Previous);  
(Dimension, Unit, Measure, Value);  
(Concept, Dimension, Unit, Measure);  
(Resource, Concept, Dimension, Unit);  
(Statement, Resource, Concept, Dimension);

Populate / align / annotate models with dimensional data. Model input: statements (model resources). Model specification: augment, sort  
statements. Model specification: specialization of base model layers. Resolve resolution statements order.

Dimensional input set model specificatíon (from Statement layer, ordered SPOs: order criteria, comparisons. Kinds / class / occurrence / instance  
order criteria?).

Value, Previous, Distance, Next. Dimension, Unit, Measure, Value (aggregated ordered statements layers).

Value -> distance(prev, next); ordering;

Assert knowledge: 1h -> 60min;

dom-lun-mar-mie-jue-vie-sab (orders);

1mt -> 100cm;

etc.

Comparison / order: Alignments (prev, curr, next asserted knowledge). Next hour, location, city, country, next distance at next time at current speed. Event sourcing / tracking: married -> marriage occurred.

Sort: cause / effect, temporal, etc. Messages align, functional map, fold, etc. Primitives. Encode layered statements ordering. Complement / supplement concepts definitions.

Events metamodel (TBD):

(Object, State, Axis, Type)  
(State, Axis, Type, Event)  
(Axis, Type, Event, Event)  
(Type, Event, Event, Event)  
(Event, Event, Event, Event)

**MetaGraph Model (models aggregations)**

See Message Resolution.

Model MetaGraph (TBD):

MetaGraph: Resource, Statement, Kind class / instance as CSPO MetaGraph statement roles. Class / subclass relationship, Kind / subkind relationship. Grammar / Model bindings.

Grammar: layers aggregate kinds from resource / statement layer or kinds for each model layers.

Layers, contexts, occurrences, kinds: Role Entity layer occurrences instantiated with each Entity SPO as Entity subject (Entities occurrences in Role context for each Entity SPO). Idem for subsequent layers.

Statement class: context.

Statement instance: context occurrence.

State resource kind in occurrence in context.

State resource (context) class / (occurrence) kind hierarchies.

State Resource URIs occurrences / class IDs.

Resolve Message matching Resource from behavior layers / matching kinds from Model / data layers.

(Kind, SuperKind, Attribute, Value);

(Occurrence, Kind, SuperKind, Attribute);

(Context, Occurrence, Kind, SuperKind); (attributes / links bindings).

(Resource, Context, Occurrence, Kind); State Resource Kind in occurrence context (context / role bindings).

(Statement, Resource, Context, Occurrence); State Resource URIs occurrences / Resource class IDs (classification bindings).

(Interaction, Statement, Resource, Context);

(Action, Interaction, Statement, Resource);

Model MetaGraph: Resource, Statement, Kind (reifying class / instances) contexts / occurrences / attributes / values.

Encoding. Message dispatch, event bus routes. URIs / IDs mappings.Resource set specification resolution. Resolve concrete resources, Message expansion. Resolve Message / dialog (CRUD) semantics.

Ontology matching (table, pk, col, val example). Helper upper models for models linking / alignment.

Events declarative definition. State change of value in axis in measure of context.

Messaging metamodel:

(Message, Resource, LHS, RHS);  
(Interaction, Message, Resource, LHS);  
(Role, Interaction, Message, Resource);  
(Context, Role, Interaction, Message);  
(Dataflow, Context, Role, Interaction);

**Datasources / Backends / Services (URIs)**

TBD.

**Addressing. IDs. Encodings**

Resource<T : URI> monadic hierarchy.

Basic hypermedia browse / CRUD (HTTP verbs) bound Message functors compatible for all Resources (REST).

Resource.flatMap(Message::apply) : Observable<Resource> (stream). Composable functions.

Basic Message application (Context Mapping): shift right mapped applied statement resources. Mapped resource context > instance (occurrence) of next layer message reified resource context.

Model MetaGraph: Resource, Statement, Kind (reifying class / instances) contexts / occurrences / attributes / values. Encoding. Message dispatch, event bus routes. URIs / IDs mappings. Resource set specification resolution. MetaGraph resolves concrete resources, Message expansion.

Resolve Message / dialog (CRUD) semantics via MetaGraph driven transforms (data / schema / behavior augmentation: dialogs).

Dimensional / Grammar models.

aX^4 + bY^3 + cZ^2 = dW;  
d, a, b, c: classes (CSPO);  
WXYZ: instances (CSPO);  
Powers: CSPO role;  
Terms: CSPO resources;  
Z(obj) is Y(pred) for X(subj) in W(ctx);

Instance, class, metaclass, occurrence terms. Primitives, variables, placeholders.

Resolution (Discovery, DIDs). Templates (grammars). Subjects: attr / val, ctx / role.

Behavior: order / compare.

Proof of work.

MetaGraph model: map URIs -> IDs.

Satisfy dW. Sync resolution (recurse terms contexts).

FCA. Resource attributes.

Tensor, adjacency matrix, tree.

Model MetaGraph: Resource, Statement, Kind (reifying class / instances) contexts / occurrences / attributes / values. Encoding. Message dispatch, event bus routes. URIs / IDs mappings.Resource set specification resolution. Resolve concrete resources, Message expansion. Resolve Message / dialog (CRUD) semantics. Dimensional / Grammar models.

Naming: Context URIs. Dimensional (Statement, Resource, Kind) addressing (conventions). Discovery: patterns / locators: Semantic URIs / MetaGraph ID mappings. Encodings: contextually encoded addresses / URIs.

Naming: NLP. Bind / suggest human readable names / labels.

Naming: Source (plain class) URIs.

Naming: Statement (Context) addresses.

Naming: Occurrence URIs (in Statement in CSPO role).

Naming: Contextually encoded addresses (URIs in Occurrences in Statements in relation with other occurrences).

Naming: Kinds addresses (global / mask, from occurrences in statements). Signatures. MetaGraph: operate over IDs.

TBD.

**Dataflow (reactive models)**

TBD.

For input Resource(s) (Model reactive / async IO APIs):  
. Create / retrieve Model  
. Create / retrieve Context Message(s)  
. Create / retrieve Message(s) Interactions  
. Bind Interaction Message Resource(s)  
. Perform Message transform. Materialize results. Message application rules: upper / domain ontology selectors (closest matching role in  
hierarchies), context alignments.

Match request statement / graph with model via context in interaction (algorithm: addressing, encoding, interaction model upper bindings /  
alignments). Resource MetaGraph. Reified model resources (Resource, Statement, Kinds, CSPO, etc.).

Apply subsequent transforms in interaction context (referrer context, get classes playing entity role, get behavior flows, browse / navigate  
streams). Context, variables, wildcards, placeholders.

Services: distributed addressing / resolution, reactive distributed event bus: streams / contracts, index, naming, registry.

Discovery: All model kinds are browseable / discoverable.

Determine class (reified layers contexts) hierarchies:

(ContextReifiedClass, ContextReifiedSubClass, SubClassAttributeKind, SubClassValueKind);

Merge / specify model, context, interaction graphs. Reified model resources, statements, kinds.

Model, context, interaction model graphs layers specifications. Reified models layers contexts resources describe graphs. Augmentation. Message context statement occurrence: Model.

Message flow (event loop) in / out:

Activation (data) <-> Alignment (schema) <-> Aggregation (behavior);

Encode behavior in statements / graph:  
Comparisons, order. Sort. Order (kinds hierarchy?).

Pattern matching, iteration, jumps. Discovery: routes / signatures, next event in bus / graph.

Context Model Message: Resource Specification (Grammar Template).

Messages Model: context model instance from input model grammar. Transform: context model instance from Messages.

Express Augmentation (Activation, Alignment, Aggregation) as Messages / Transforms. Reified Model entity types / roles (CSPO, Kinds, Layers, etc.).

Resource monad of URIs or Message monad of Resource?

Encoding. Addressing. Schema / MetaModel for data (Model), schema (Context), behavior (Interaction) resources / layers (aggregation). Naming  
formats / schemes: namespaces, contexts.

Class hierarchies (express context / class / kinds hierarchy). Grammars / Dimensional metadata.

Resource MetaGraph bindings (Message expansion / resolution index).

Subscription, Subscriber, Producer, Consumer, Processor. Example: submitting Behavior layer grammar / context "template" initiates "dialog" for fulfill Behavior expanding Message(s) and nested context layer statements (known / resolvable, new behavior / subitems) needed to complete / update full Behavior layers contexts graph.  
Augment. Alignment, Activation, Aggregation Message(s) : Resource set specifications.

Model listens onMessage (interaction context model population / dialogs scopes / namespaces).

Model augments input Message (augmentation specifications over in Message).

Model expands Message (Message over model resources):

Resource listen modelMessage. Model subscribes to response.

Matching triggered Resource. Message matching semantics (transforms).

Triggered Resource publish itself modelMessage.

Model augments output Message (augmentation specifications over out Message).

Model publish onMessage (interaction context model dialogs / resource dumps).

**Messages: Transforms. Graph Execution Semantics**

Message encoding semantics resolve transform execution resource set declaratively from MetaGraph / Models.

Specification resolves to query / create / update / delete according interaction contexts. Messages models determines “possible” messages according models grammars. Interaction specifications (statement / graph / dialog) may have any message encoding components in corresponding statement roles.

For each behavior, flow, class, kind, entity, statement in input request, transforms matches those components by applying messages into model resources (grammar) matched into interaction model (binding subsequent roles by dialogs).

New (potentially unknown) resources are added and augmented into the graph. Augmented resource events emitted from transform streams.

Example: a message composed of a kinds CSPO matches statements “instances” of those specifications (statements whose CSPO have matching kinds). A message with three CSP kinds and a (potentially unknown) object URI retrieves matching resources having that object value into corresponding property kinds. An statement of plain (potentially unknown) URIs instantiates / updates and augments new / known resources added to models and returns an augmentation transform result.

Interaction Model: Context of Messages model for a given interactions session / dialog state. Message invocation requests: Statement(s) building Resource invocation graph with layers matching Message patterns. Layers graph invocation patterns matching from higher to lower layers resources fulfilling higher layers templates. Variables, wildcards, placeholders.

Dialog arguments resolutions example: higher layer Resource / Message request / invocation instantiates in Interaction Transform context corresponding lower layer graph statements to be “populated” to fulfill request. Message IO of “forms” (Messages) inter-peers (originating peer  
acting as “server”) for initial requested peer to “ask” for form elements to be populated (interaction context “dialogs”). Resolution may propagate to other peers (content aware addressing dataflow routes dispatch: P2P resources address encodings, matching forms models requests). Nested interactions.

Explain messages (resource resolution). Grammar. Match model Resource(s). Compound nested CSPO statement contexts defines result behaviors. Message CSPO contexts may define create, retrieve, update or delete operations (passing 'null' for example for resource / statement to be deleted).

Explain transforms (message application). Transform: Resource stream result of Message application over resolved Resource(s)). Input statements: Message(s) / Resource(s) (from input message or to be populated or populated in dialog) and "goal" Message / Resource aggregating a model from Resource MetaGraph with Message / Resource bindings.

Message types (Augmentation: onto / domains):  
Attribute / Link (data):  
. Alignment: Augment / infer Attribute / Link.  
Class / ID (schema):  
. Activation: Augment / infer Kind, Class.  
Role / Context (behavior):  
. Aggregation: Augment / infer Role / Context.

Runtime / Resources / Messages: Core (upper / onto) Resources, Messages, Transforms. Reified entities (CSPO, Kind, SubjectKind, etc.). Match cases in messages.

Core (upper / onto) Messages: Getters, setters, nav, etc.

Domain Messages: raiseSal: setSal(sal \* increment); promotion: setPosition.

Event sourcing / tracking: married -> marriage occurred.

Resource.flatMap(messageInst::apply) : Resource.

Dataflow: Messages hierarchy. Aggregate contexts from coarse to fine grained transforms (raiseSal -> setAttr).  
data <-> schema <-> behavior.

Message dispatch, input statements resolve to applicable messages from switch from behavior to data layer invoking async microservice.  
Message case matching may involve entering and leaving data, schema and behavior paths if aggregated contexts matches more than one  
message. Visitor.

Message: functor (monadic transform) : Resource<T> -> R, T, R : URIs (hierarchies, models, semantic content types). Available verbs / flows /  
navigation (browse models, state of application returned from materialized models). Parameterized functions (partial applications) into Messages metamodel resources. Contexts (dataflow). Execution graph.

Alignment Message: Resource -> Statements (attributes, values).  
Activation Message: Statement -> Kind, Class.  
Aggregation Message: Statement -> Statement (next layer).

Subscriptions declarations / definitions. Applied on streams activations (transforms, executions resource parameterized partial contexts).

Messages metamodel: functor declarations partially defined over metamodels resource (T) defining transforms into (R) over application  
(flatMap) over / into (S). Messages inferred / aligned, activated, aggregated according base message transforms resources. Messages inferred from models / layers. TBD.

Functors <T, R> -> Resource<R>

Form / Template describing (reified as a Resource in a context model) declaratively subscriptions and actual exchange capabilities (datflow).  
Mappings, Transforms.

Processor which acts upon Resource events. Materialize results.

Specify declaratively augmentations by means of messages.

Upper onto / domain aggregated messages.

Event bus: P2P deployment.

Messages: Monadic applicables over Resource (flatMap).

Base HTTP / Browse (REST) Messages. Custom Messages.

Model MetaGraph: Resource, Statement, Kind (reifying class / instances) contexts / occurrences / attributes / values. Encoding. Message dispatch, event bus routes. URIs / IDs mappings. Resource set specification resolution. Resolve concrete resources, Message expansion. Resolve Message / dialog (CRUD) semantics.

**Augmentation (via Messages)**

Activation (Statement / Entities : data).

Alignment (Kinds / Classes : context / schema).

Aggregation (Flows / Behaviors : interaction).

Messages describes declaratively augmentation steps materializing models contexts / hierarchy layers.

**Protocol (API): dialogs (distributed resource augmentation / sync)**

Message resolution (contexts).

Reactive. Interaction / session contexts.

Annotate, link, browse resources instances, classes, metaclasses, occurrences in roles in contexts, attributes / values. Services / clients: endpoints: Virtualization (wrapper protocols).

Semantically annotated content types: image/png;face, text/xml;faceImgCoords. RDF schemas describing content, attributes, links in context / target roles. Content types: labels (schemas).

Message: Context Model API. Input statements: Model Grammar. Augmented IO by interaction transforms of applied matching Message with model statements inputs. Context of core models instances. API.

Transform: Interaction Model API. Input statements: Transform request invocation specification. Functional application of Message(s) over Resource(s): Transform (streams). Augmented IO: Requested Transform which applied augments resulting responses (dialog arguments  
resolutions). Context of context model instances.

Reactive / streams API. Message Transform (interaction result): matches request context specification built upon Resources / Messages (TransformBuilder). Resolve state / dialog session graph. Returns observable stream. Dataflow (chaining). Operations (over streams).

Transform request invocation specifications: means to interact with underlying contexts models (CRUD, domains behavior). Transforms result from applicating Message(s) over Resource(s). Sending a Message Resource to a given interaction context initiates a “dialog” in which to “populate” target Resource(s) and Resource arguments. Each dialog “step” renders resources / layers streams of requested arguments (server “queries” clients) or resources / layers streams of response augmented Resource(s).

Graph linking / alignment / synchronization by entailments from event sourcing over inferred state. Distributed predictive alignments.

DOM / OGM APIs (JAF). I/O Implementation, Deployment.  
Model, URI, Resource, Statement, Kind hierarchies. Models architecture (URI class per layer). DIDs / P2P / Rx Implementations. Model API. ModelManager. Event loop. IO.

**Protocol (API): resource activation (hypermedia application browser)**

Reactive. Interaction / session contexts.

Protocols / Services / Clients: Context interaction sessions (state flows).

Content type activation. Messages / gestures. Rules (commands / verbs). Content types: labels (schemas).

Browser referring context (Work, Peter, Employee).

Annotations (protocol): JSON-LD. Model / Grammar / Dimensional. Map annotations to resources (query string / meta resource description). Browse data (model), schema (grammar), behavior (metagraph).

Models ‘plug’ into Runtime augmenting its capabilities via standard extension APIs (added features / knowledge reactive URIs). Models ‘modules’: parsing modules declarative descriptions. Augment, link instance data.

Upper aligned ontology plugins / blueprints:

Resource URIs specialized implementations for different connectors / endpoints and content types (DB / OData, REST / HAL, etc.). Feature  
Resources backends (i.e.: URI for DB interaction).  
Purposes: Metamodel declarative goal statement. Fulfill flows (templates / forms: Messages).

Goal: P2P service that connects to services / endpoints (DB, REST, etc.), homogenizes them and exposes an API by which (augmented)  
knowledge of an stated entity is returned in response (protocol that entails queries / CRUD, object navigation in message / session state contexts). Peer shares / syncs with other peers.

Goal: Intermediate API (HAL for example) aggregating previous objects knowledge (DCI, DOM, OGM, MVC)

Goal: Semantic Browser. Homogenize diverse domains. Query examples. Search session history. Referrer semantics. Collected items in goals roles. Create session purpose document. Link to / from any addressable resource in context / role. Annotate source / destination context roles,  
attributes and schema.

TBD.

**Ontology matching**

TBD.

NLP NER.

CEP:

Cause / Effect.

Periodical (units lengths).

Order inference.

Hierarchies. Containment (Graph).

Dimensions (axis / measures).

Facts. Relationships (events, assertions).

**Data / Reference Model (APIs, Functional Semantics)**

TBD.

Upper ontology: Node "levels" of domains abstraction. Highest level: service / user interaction (resource / hypermedia activation: model gestures). Lowest levels: upper ontology / business domains.

Application / Site / Service node types (Node ontologies domains layers). Renderers producers / consumers. Backends integration (Augmentation, Messages).

**Platform: implementation**

Introduction. Document. Use Cases (EHR). Standards. Models (predictions / signatures).

Implementation. Languages. Backends. Reactive frameworks / microservices. Distributed consistency. P2P / DIDs. Models / APIs. Nodes / Endpoints. Containers. Deployment.

Implementation: render RDFS / OWL upper ontology aligned (sameAs, type, subClassOf, restrictions, etc.). from Model / Message+ XSLT transforms. Semantic engine / reasoner / backend (URI published reactive service, Message based wrapper). Record Model / Message transforms.

API: URI, Resource, Message, Statement, Kind, Layers. Representation: XML bindings.

Kind : Statement : Message : Resource : URI;

URI<T extends URI> : Monad.

Resource: (URI, URI, URI, URI);

Message: specification / transform (input / output dialog).

XSLT / XPath / XLink / XPointer / XQuery.

Resource XML Encoding (nested layers quads).

Message XML Encoding.

XSLT templates (Resolution, Activation, Alignment, Aggregation). Resolution algorithm: TBD (ontology matching).

Events: Dataflow. Reactive Model endpoint Message dispatch / resolution (Producer). Resolve (addressable) Message resources (Resolution template). Apply templates (Resolved resources : model / Message resources : view context) : XML (Message).

Ontology levels: data / schema / behavior (backend, business, frontend) objects.

TBD.

**Messages: Transforms. Graph Execution Semantics**

Message encoding semantics resolve transform execution resource set declaratively from MetaGraph / Models.

Specification resolves to query / create / update / delete according interaction contexts. Messages models determines “possible” messages according models grammars. Interaction specifications (statement / graph / dialog) may have any message encoding components in corresponding statement roles.

For each behavior, flow, class, kind, entity, statement in input request, transforms matches those components by applying messages into model resources (grammar) matched into interaction model (binding subsequent roles by dialogs).

New (potentially unknown) resources are added and augmented into the graph. Augmented resource events emitted from transform streams.

Example: a message composed of a kinds CSPO matches statements “instances” of those specifications (statements whose CSPO have matching kinds). A message with three CSP kinds and a (potentially unknown) object URI retrieves matching resources having that object value into corresponding property kinds. An statement of plain (potentially unknown) URIs instantiates / updates and augments new / known resources added to models and returns an augmentation transform result.

Interaction Model: Context of Messages model for a given interactions session / dialog state. Message invocation requests: Statement(s) building Resource invocation graph with layers matching Message patterns. Layers graph invocation patterns matching from higher to lower layers resources fulfilling higher layers templates. Variables, wildcards, placeholders.

Dialog arguments resolutions example: higher layer Resource / Message request / invocation instantiates in Interaction Transform context corresponding lower layer graph statements to be “populated” to fulfill request. Message IO of “forms” (Messages) inter-peers (originating peer  
acting as “server”) for initial requested peer to “ask” for form elements to be populated (interaction context “dialogs”). Resolution may propagate to other peers (content aware addressing dataflow routes dispatch: P2P resources address encodings, matching forms models requests). Nested interactions.

Explain messages (resource resolution). Grammar. Match model Resource(s). Compound nested CSPO statement contexts defines result behaviors. Message CSPO contexts may define create, retrieve, update or delete operations (passing 'null' for example for resource / statement to be deleted).

Explain transforms (message application). Transform: Resource stream result of Message application over resolved Resource(s)). Input statements: Message(s) / Resource(s) (from input message or to be populated or populated in dialog) and "goal" Message / Resource aggregating a model from Resource MetaGraph with Message / Resource bindings.

Message types (Augmentation: onto / domains):  
Attribute / Link (data):  
. Alignment: Augment / infer Attribute / Link.  
Class / ID (schema):  
. Activation: Augment / infer Kind, Class.  
Role / Context (behavior):  
. Aggregation: Augment / infer Role / Context.

Runtime / Resources / Messages: Core (upper / onto) Resources, Messages, Transforms. Reified entities (CSPO, Kind, SubjectKind, etc.). Match cases in messages.

Core (upper / onto) Messages: Getters, setters, nav, etc.

Domain Messages: raiseSal: setSal(sal \* increment); promotion: setPosition.

Event sourcing / tracking: married -> marriage occurred.

Resource.flatMap(messageInst::apply) : Resource.

Dataflow: Messages hierarchy. Aggregate contexts from coarse to fine grained transforms (raiseSal -> setAttr).  
data <-> schema <-> behavior.

Message dispatch, input statements resolve to applicable messages from switch from behavior to data layer invoking async microservice.  
Message case matching may involve entering and leaving data, schema and behavior paths if aggregated contexts matches more than one  
message. Visitor.

Message: functor (monadic transform) : Resource<T> -> R, T, R : URIs (hierarchies, models, semantic content types). Available verbs / flows /  
navigation (browse models, state of application returned from materialized models). Parameterized functions (partial applications) into Messages metamodel resources. Contexts (dataflow). Execution graph.

Alignment Message: Resource -> Statements (attributes, values).  
Activation Message: Statement -> Kind, Class.  
Aggregation Message: Statement -> Statement (next layer).

Subscriptions declarations / definitions. Applied on streams activations (transforms, executions resource parameterized partial contexts).

Messages metamodel: functor declarations partially defined over metamodels resource (T) defining transforms into (R) over application  
(flatMap) over / into (S). Messages inferred / aligned, activated, aggregated according base message transforms resources. Messages inferred from models / layers. TBD.

Functors <T, R> -> Resource<R>

Form / Template describing (reified as a Resource in a context model) declaratively subscriptions and actual exchange capabilities (datflow).  
Mappings, Transforms.

Processor which acts upon Resource events. Materialize results.

Specify declaratively augmentations by means of messages.

Upper onto / domain aggregated messages.

Event bus: P2P deployment.

Messages: Monadic applicables over Resource (flatMap).

Base HTTP / Browse (REST) Messages. Custom Messages.

Model MetaGraph: Resource, Statement, Kind (reifying class / instances) contexts / occurrences / attributes / values. Encoding. Message dispatch, event bus routes. URIs / IDs mappings. Resource set specification resolution. Resolve concrete resources, Message expansion. Resolve Message / dialog (CRUD) semantics.

**Augmentation (via Messages)**

Activation (Statement / Entities : data).

Alignment (Kinds / Classes : context / schema).

Aggregation (Flows / Behaviors : interaction).

Messages describes declaratively augmentation steps materializing models contexts / hierarchy layers.

**Protocol (API): dialogs (distributed resource augmentation / sync)**

Message resolution (contexts).

Reactive. Interaction / session contexts.

Annotate, link, browse resources instances, classes, metaclasses, occurrences in roles in contexts, attributes / values. Services / clients: endpoints: Virtualization (wrapper protocols).

Semantically annotated content types: image/png;face, text/xml;faceImgCoords. RDF schemas describing content, attributes, links in context / target roles. Content types: labels (schemas).

Message: Context Model API. Input statements: Model Grammar. Augmented IO by interaction transforms of applied matching Message with model statements inputs. Context of core models instances. API.

Transform: Interaction Model API. Input statements: Transform request invocation specification. Functional application of Message(s) over Resource(s): Transform (streams). Augmented IO: Requested Transform which applied augments resulting responses (dialog arguments  
resolutions). Context of context model instances.

Reactive / streams API. Message Transform (interaction result): matches request context specification built upon Resources / Messages (TransformBuilder). Resolve state / dialog session graph. Returns observable stream. Dataflow (chaining). Operations (over streams).

Transform request invocation specifications: means to interact with underlying contexts models (CRUD, domains behavior). Transforms result from applicating Message(s) over Resource(s). Sending a Message Resource to a given interaction context initiates a “dialog” in which to “populate” target Resource(s) and Resource arguments. Each dialog “step” renders resources / layers streams of requested arguments (server “queries” clients) or resources / layers streams of response augmented Resource(s).

Graph linking / alignment / synchronization by entailments from event sourcing over inferred state. Distributed predictive alignments.

DOM / OGM APIs (JAF). I/O Implementation, Deployment.  
Model, URI, Resource, Statement, Kind hierarchies. Models architecture (URI class per layer). DIDs / P2P / Rx Implementations. Model API. ModelManager. Event loop. IO.

**Protocol (API): resource activation (hypermedia application browser)**

Reactive. Interaction / session contexts.

Protocols / Services / Clients: Context interaction sessions (state flows).

Content type activation. Messages / gestures. Rules (commands / verbs). Content types: labels (schemas).

Browser referring context (Work, Peter, Employee).

Annotations (protocol): JSON-LD. Model / Grammar / Dimensional. Map annotations to resources (query string / meta resource description). Browse data (model), schema (grammar), behavior (metagraph).

Models ‘plug’ into Runtime augmenting its capabilities via standard extension APIs (added features / knowledge reactive URIs). Models ‘modules’: parsing modules declarative descriptions. Augment, link instance data.

Upper aligned ontology plugins / blueprints:

Resource URIs specialized implementations for different connectors / endpoints and content types (DB / OData, REST / HAL, etc.). Feature  
Resources backends (i.e.: URI for DB interaction).  
Purposes: Metamodel declarative goal statement. Fulfill flows (templates / forms: Messages).

Goal: P2P service that connects to services / endpoints (DB, REST, etc.), homogenizes them and exposes an API by which (augmented)  
knowledge of an stated entity is returned in response (protocol that entails queries / CRUD, object navigation in message / session state contexts). Peer shares / syncs with other peers.

Goal: Intermediate API (HAL for example) aggregating previous objects knowledge (DCI, DOM, OGM, MVC)

Goal: Semantic Browser. Homogenize diverse domains. Query examples. Search session history. Referrer semantics. Collected items in goals roles. Create session purpose document. Link to / from any addressable resource in context / role. Annotate source / destination context roles,  
attributes and schema.

TBD.

**Ontology matching**

TBD.

NLP NER.

CEP:

Cause / Effect.

Periodical (units lengths).

Order inference.

Hierarchies. Containment (Graph).

Dimensions (axis / measures).

Facts. Relationships (events, assertions).

**Data / Reference Model (APIs, Functional Semantics)**

TBD.

Upper ontology: Node "levels" of domains abstraction. Highest level: service / user interaction (resource / hypermedia activation: model gestures). Lowest levels: upper ontology / business domains.

Application / Site / Service node types (Node ontologies domains layers). Renderers producers / consumers. Backends integration (Augmentation, Messages).

**Platform: implementation**

Introduction. Document. Use Cases (EHR). Standards. Models (predictions / signatures).

Implementation. Languages. Backends. Reactive frameworks / microservices. Distributed consistency. P2P / DIDs. Models / APIs. Nodes / Endpoints. Containers. Deployment.

Implementation: render RDFS / OWL upper ontology aligned (sameAs, type, subClassOf, restrictions, etc.). from Model / Message+ XSLT transforms. Semantic engine / reasoner / backend (URI published reactive service, Message based wrapper). Record Model / Message transforms.

API: URI, Resource, Message, Statement, Kind, Layers. Representation: XML bindings.

Kind : Statement : Message : Resource : URI;

URI<T extends URI> : Monad.

Resource: (URI, URI, URI, URI);

Message: specification / transform (input / output dialog).

XSLT / XPath / XLink / XPointer / XQuery.

Resource XML Encoding (nested layers quads).

Message XML Encoding.

XSLT templates (Resolution, Activation, Alignment, Aggregation). Resolution algorithm: TBD (ontology matching).

Events: Dataflow. Reactive Model endpoint Message dispatch / resolution (Producer). Resolve (addressable) Message resources (Resolution template). Apply templates (Resolved resources : model / Message resources : view context) : XML (Message).

Ontology levels: data / schema / behavior (backend, business, frontend) objects.

TBD.