**Implementation:**

Definitions: Quads, contexts, Kinds, Grammar, etc.

Implementation: Runtime. Architecture. Components. Patterns. Models. Messages. Augmentation. Events dispatch. Message aggregation / instantiation / resolution / application. Backends. Services.

Resources inputs / outputs: Augmentation, Protocol, Browser. Message addressing / resolution / application.

Component: Models (data). Source: augmented input statements. APIs (Model).

Component: Messages (contexts). Source: augmented models templates (Grammar). APIs (Model).

Component: Transforms (interactions): Source: input statements case matching Message inputs. Returns / materialize results. APIs (Model).

Core Model API: Augmented (Aligned, Activated, Aggregated inputs matching model context messages) IO. Resource MetaGraph. Dimensional model. Grammar. Model repository. Backend. API.

Core API: Model, URI, Resource, Statement, Kind.

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).

Message IO encoding components kinds:

Data: Assertion (statement / entity).

Schema: Type (kind / class).

Behavior: Interaction (flows / behaviors).

Specification resolves to query / create / update / delete according interaction contexts. Messages models determines “possible” messages according models grammars. Interaction specification (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 subsecuent roles by dialogs). New (potentially unknown) resources are added and augmented into the graph. Augmented resource events emited 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 inititial 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.

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 subsecuent 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.

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);

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. Resolve concrete resources, Message expansion. Resolve Message / dialog (CRUD) semantics.

///

**Purpose driven hypermedia activation:**

Protocols / Services / Clients: Context interaction sessions (state flows).

Content type activation. Messages / gestures. Rules (commands / verbs).

Browser referring context (Work, Peter, Employee).

Models ‘plug’ into Runtime augmenting its capabilities via standard extension APIs (added features / knowledge).

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.

///

**Resource Activation Protocol**

Annotate, link, browse resources instances, classes, metaclasses, occurrencesin roles in contexts. 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.

///

**Messages Metamodel (Context Model):**

Explain models (resources, statements, kinds).

Explain layers / aggregation.

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 appplication). 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. Applyied on streams activations (transforms, executions resource parameterized partial contexts).

Messages metamodel: functor declarations partially defined over metamodels resource (T) defining transforms into (R) over appplication (flatMap) over / into (S). Messages inferred / aligned, activated, aggregated according base message transofrms 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.

///

**Addressing: URIs, DIDs URLs. Adddress, content type, representation (URI APIs). Browse / CRUD (DAV).**

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 of mesage reified resource context.

Transforms (Message templates):

Resource: Statement

Message: SubjectKind

Transform: Resource

Resource: Statement

Message: ClassLayer

Transform: Statement (class)

Resource: Employee

Message: Position

Transform: Manager

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.

///

**Resource API**

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 service 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.

///

**Quads reference model. Kinds. Grammar.**

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. TBD.

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.

///

**Metamodels: Graphs (Models)**

Composed of quads semantically aggregated into layers.

Core features provides:

Alignment

Activation

Aggregation

Message / Transform driven specification of Alignment, Activation, Aggregation.

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. TBD.

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?

///

**Encodings**

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, ocurrence terms.

primitives, variables, placeholders.

resolution (Discovery, DIDs).

Templates (grammar)

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.

///

**Reactive Dataflow. Resource / Message / Transform. Behavior in graphs.**

Message flow (event loop) in / out: Alignment (data) <-> Activation (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 (Alignment, Activation, 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). 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).

///

**DCI Metamodel (Base Model)**

(Resource, Resource, Resource, Resource) : Resource (Model).

(Statement, Subject, Attribute, Value);

(Role, Statement, Attribute, Value);

(Kind, Role, Statement, Attribute);

(Class, Kind, Role, Statement);

(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. Resolve concrete resources, Message expansion. Resolve Message / dialog (CRUD) semantics.

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);

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 layers).

///

**Dimensional Metamodel**

(Value, Distance, Previous, Next);

(Measure, Value, Distance, Previous);

(Unit, Measure, Value, Distance);

(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, lun-mar-mie-jue-vie, 1mt -> 100cm.

Comparison / order: Alignments (prevv, curr, next). 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 / suplement 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)

Model MetaGraph (TBD):

(ResourceClass, ResourceID, Statement, Kind);

(StatementClass, StatementID, ResourceID, Kind);

(KindClass, Kind, StatementID, Kind);

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);

///

**Event sourcing (“offline” sync). API**

Graph linking / alignment / sinchronization by entailments from event sourcing over inferred state.

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: Input statements for querying augmented knowledge. Browse result model graphs. Input statements encoding queries / commands: grammars, reified message contexts (templates / forms). Browseable models, contexts, interactions (state / content semantic activation). Dataflow according Messages input signatures.

**Meta Model:**

Meta Model: Layered class hierarchy of RDF Statement (Quads) stacked Layers Mappings. Model shapes / grammar metadata. Augmentations reify to / from meta model mappings / CSPO ontology statements.

Layers: URI, Instance, Class, Context, Metaclass, Occurrence, Statement, Relation, Relationship shape mappings (layers transforms / message augmentation templates) grammar shapes.

Model Layers rendered into different APIs / Representations (RDF4J SAIL Layers):

RDF Store SAIL:

Quads Layers / Objects Mappings Meta Model Metadata Encoding: RDF / RDFS Ontology (Upper / Schema). Meta Model Reified Order / Matching Encoding.

Object / Graph Mapper (OGM) SAIL:

Object Mappings: Singleton POJOs mapping Quads Statements plus helper methods (aggregate context subjects, subject attributes, etc.). Quads / Functional IO. Layers Object Mapping (Meta Model metadata populated).

Functional APIs SAIL:

Functional Layers Wrapper: Augmentations, Metadata, Service Transforms Namespaces. Meta Model Reified Functions / Mappings Encoding.

Facades SAIL:

Business Layer: DCI Ontology Models for Forms / Flows (HAL / HATEOAS) Use Cases rendering: Declarative REST / WS-\* Endpoint Templates. Application Ontology aligned Models.

Layers specs. All SAILs renders in some form the following data model / schema:

URI

(URI, URI, URI, URI)

Instance : URI

(Instance, URI, URI, URI)

Class : Instance

(Class, Instance, URI, URI)

Context : Class

(Context, Class, Instance, URI)

Metaclass (Kind) : Context

(Metaclass, Context, Class, Instance)

Occurrence : Metaclass

(Occurrence, Metaclass, Context, Class)

Statement (Entity) : Occurrence

(Statement, Occurrence, Metaclass, Context)

Relation : Statement

(Relation, Statement, Occurrence, Metaclass)

Relationship : Relation

(Relationship, Relation, Statement, Occurrence)

Aligned Statements. Entity ontology matching / alignment. Multiple Statement Occurrences. TBD.

Augmentations:

Layers Aggregation:

Determine (Layers) which Subjects (Occurrences, aMarriage : Relation) belongs / aggregated into which Statements (Contexts, marriage : Relationship).

Layers Activation:

Determine Occurrence Attributes.

Layers Alignment:

Determine Occurrence Attributes Values.

Meta Model:

Map Layers to CSPO inter / intra Roles. Model / Meta Model Mappings.

Message (SAIL Statements IO) to / from Meta Model:

(C, S, P, O);

to / from:

(Context / Statement, Object / Occurrence, Attribute / Metaclass, Value / Context); (Statement, Relation, Relationship).

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);

Events Flow:

URI Layer: Input Layer. Message populates URI Layer.

URI Layer: Augmentations Layers reified (levels). Augmentations Performed.

URI Layer: Outputs materialized URI Statements from query Augmentations final Relationships. Return matching / augmented Message statements.

Layers Meta Model schema:

RDF Store SAIL.

TBD.

Objects Layers Mappings:

Object / Graph Mapper (OGM) SAIL.

TBD.

Functional Layers Mappings:

Functional APIs SAIL. Perform Augmentations / Functional Transforms.

Functional Dataflow.

Layer Monad. Augments Object Mapping for Functional Transforms.

Functional Transforms (Augmentations, Metadata, Services namespaces).

Message : Set<Statement : URI>.

Layer<Set<URI>>

Layer<Set<Instance>>

Layer<Set<Class>>

Layer<Set<Context>>

Layer<Set<Metaclass>>

Layer<Set<Occurrence>>

Layer<Set<Statement>>

Layer<Set<Relation>>

Layer<Set<Relationship>>

Transforms:

For each dataflow bindings (parameterized types):

Augmentations::Aggregate<T, U>

Augmentations::Activate<U, V>

Augmentations::Align<V, W> : Materialized Augmentation Statements.

Metadata namespace: Model Browsing Transform Functions:

Metadata::URIs<T, U>

Metadata::Instances<T, U>

Metadata::Classes<T, U>

Metadata::Contexts<T, U>

Metadata::Metaclasses<T, U>

Metadata::Occurrences<T, U>

Metadata::Statements<T, U>

Metadata::Relations<T, U>

Metadata::Relationships<T, U>

Metadata transforms (browse: order / filter / traversal). Example: Filter, obtain Address Kind Occurrence in Person Statement Entity.

Layer<Set<Statement>> aPerson = Layer.of(personStmts);

aPerson.flatMap(Metadata::Metaclasses).stream().filter(addressKindPredicate).flatMap(Metadata::Occurrences).stream();

TBD.

Services Context Namespaces:

Function / Mapping / Service IO resolver Transform (reactive / events: dataflows). Resolve / Augment Layer occurrences. Context Resolution / matching: Shapes / Zippers (Monads).

Services::Resolve<T, U>

TBD.

Order:

Meta Model: Layers Instances Order. Metadata browse / traversal transforms results order (in context). Layer Alignment. URIs Comparator. Metadata transform (example):

(Instance, URI : Ctx, URI : LT, URI : GT)

TBD.

Matching / Alignment:

TBD.

TBD: Order / Matching / Mappings / Functions / Business: Meta Model Encoded Upper Dimensional Model. Declarative. Templates. Shapes. Addressing (Zippers).

Example:

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

Dimension:

Unit:

Measure : Event (Previous Value state). Mapping / Function argument.

Value: Current Value state (Previous Measure). Mapping / Function result.

Equals: (Measure, Value) -> (Value, Measure).

Examples:

(Time, DayOfWeek, Sunday, Monday);

(Time, DayOfWeek, Monday, Tuesday);

(Time, DayOfWeek, Tuesday, Wednesday);

(Time, DayOfWeek, Wednesday, Thursday);

(Time, DayOfWeek, Thursday, Friday);

(Time, DayOfWeek, Friday, Saturday);

Mappings for: Today, Yesterday, Tomorrow. TBD. DayOfWeek: Ordering Relation Mapping / Function ontology kind.

(Employment, Status, Unemployed, Employee);

(Employment, Status, Employee, Retired);

Mappings for Hiring / Retirement. TBD.

Resolver Service: Resolver.resolve(URI : Layer encoded DIDs) : Aligned, dereferenceable Resource (URI : Resource). Index (VSM / FCA), Naming (Addressing), Registry (Hierarchical Key / Value) Alignment Services.

Ontology Matching: OntResource. DIDs. Blockchain (updates): Endpoints (LOD / sameAs like context resources / arcs assertions / metadata). Augmentation transactions (Resource occurrences blocks). Smart Contracts / Behavior (REST / HAL DCI APIs) Metadata.

To Do: Aggregate SPO statements into (to / from) Relation / Relationship via metadata properties. Order alignment: dimensional / reified schema. Upper alignments: master / detail, cause / effect, temporal, etc.

Resources: Mappings (Relations / Functions from Resource occurrences state) Monads / Transforms: Query / Traversal.

Mappings Properties (Resource occurrences state): metaclass, class, instance, occurrence, role, context metadata properties.

Sets Aggregation: Contexts. Kinds, Resources Statement Monad.

Monad::flatMap(Monad.value);

Context Mappings: occurrence / context metadata:

Statement.of(Context);

* Occurrence: Mapping.
* Context: Attribute / Value (single map argument: retrieve values from occurrence resource mapping).
* S::flatMap(P, O) : C;
* P::flatMap(S, O) : C;
* O::flatMap(S, P) : C;

Kinds Mappings: metaclass / class / role metadata:

Statement.of(Kind);

* Role: Mapping.
* Class: Context.
* Metaclass: Kind.
* S::flatMap(C) : SK;
* P::flatMap(C) : PK;
* O::flatMap(C) : OK;

Resource Mappings: instance metadata:

Statement.of(URI); Layers Resource hierarchy.

* Instance: Mapping.
* Attribute / Value Mappings.
* S::flatMap(P) : O;
* S::flatMap(O) : P;
* P::flatMap(S) : O;
* P::flatMap(O) : S;
* O::flatMap(S) : P;
* O::flatMap(P) : S;
* Context(Kind) : Resource;
* Context(Resource) : Kind;
* Resource(Kind) : Context;
* Resource(Context) : Kind;
* Kind(Context) : Resource;
* Kind(Resource) : Context;

Example (occurrences navigation)::

Kind(Resource) : Context; Kind occurrences coming from previous Resource (navigation context).

Sets, Groups, Categories:

Sets: Contexts, Kinds, Roles, Dimensions.

Individuals: Relations, Relationships, Resources, Dimensions.

OntResource Monad:

Aligned Entities / Resolver Service.

OntResource<Dimension>;

OntResource<Role / Resource>;

OntResource<Relationship / Kind>;

OntResource<Relation / Context>;

Occurrences:

Resource<?>.flatMap(Resource::occurrences) : Resource<Set<Statement>>;

Statement: Context / Relation Dimensional Mappings / Alignments Statements.

Statement Monad: Occurrences

Statement.of(Contexts, Kinds, Roles, Dimensions);

Statement.of(Relations, Relationships, Resources, Dimensions);

Statement.of(Types: Dimension, Role / Resource, Relationship / Kind, Relation / Context) : Set<Statement<Type>>;

Statement::occurrences

Dimension occurrences: Resources.

Resource occurrences: Kinds.

Kind occurrences: Contexts.

(Dimension, Resource, Kind, Context);

Example:

Kind(Resource) : Context; Kind occurrences coming from previous Resource (navigation context).

Dimension occurrences: Roles.

Role occurrences: Relationships.

Relationship occurrences: Relations.

(Dimension, Role, Relationship, Relation);

Mappings / Alignments:

Dimension: Layers. Hierarchy. Encoding (sets / instances). Types. Meta Model hierarchies. Encode / align Sets / Individuals: TBD.

(Dimension, Role, Resource);

(Dimension, Kind, Relationship);

(Dimension, Context, Relation);

(Dimension, Role : Measure, Relationship : Unit, Relation : Value);

(Dimension, Resource : Measure, Kind : Unit, Context : Value);

Mappings / Transforms / Relations / Matching: Dimensional reified. Alignments (upper / matching / order / aggregation / comparison metadata).

Dimensional mappings / navigation context for occurrences : Dimension mappings hierarchy root.

Resource (URN) / Role (CSPO).

Relationship (Employment) / Kind (employee / employer).

Relation (anEmployment) / Context (salary / position).

Relation values in Contexts (roles): Data / Interactions. Sets, FCA, VSM. Base SAIL Layer. OGM, Functional Layers.

Augmentations:

Contexts: sets, Occurrences: individuals, Mappings: Contexts Occurrences.

Aggregation: Clustering (Types).

Activation: Classification (Attributes in Type Context).

Alignment: Regression (Values in Attributes Context).

Contexts (sets):

(Relation, Relationship, Role, Dimension);

Occurrences (individuals):

(Context, Kind, Resource, Dimension);

Mappings (set / individual relationships):

(Context : Dimension, Occurrence : Measure, Attribute : Unit, Value : Value);

Contexts, Occurrences, Mappings: Quads layers hierarchies. Augmentations: Aggregation, Activation, Alignment. Reification: roles / polymorphism (layers contexts type hierarchy).

Contexts:

Dimension : Mapping Context

Role, Dimension : Mapping Occurrence

Relationship, Role, Dimension : Mapping Attribute

Relation, Relationship, Role, Dimension : Mapping Value

Occurrences:

Dimension : Mapping Context

Resource, Dimension : Mapping Occurrence

Kind, Resource, Dimension : Mapping Attribute

Context, Kind, Resource, Dimension : Mapping Value

Mappings:

Value

Attribute, Value

Occurrence, Attribute, Value

Context, Occurrence, Attribute, Value

Mapping Contexts / Occurrences / Attributes / Values: Dimensions, Sets, Individuals: (Mapping.of(...)). CSPO roles : Mapping roles according Contexts. Transforms: occurrences, attributes, values, aggregate, activate, align : Mapping<?>.

Example:

Kind(Context : Resource) : Context; Kind occurrences coming from previous Resource (navigation context).

Example Mapping:

(Context: SomeDimension, Occurrence: anIndividual Occurrence Resource Measure, Attributes, Values);

Set / Instance Mapping Contexts, Occurrences, Attributes, Values. Mapping Role types (Attributes / Values) for each Context / Occurrence types.

Render Mappings to / from Contexts / Occurrences / SPOs.

Augment Relationship Relations / Relation Relationships: Mapping, Statement, Resource transforms. Dimensional / Upper Augmentations. Reify / materialize to / from SPO Statements.

**Items:**

Implementation: RDF4J Core Spring / CDI configured.

MetaModel core RDFS schema / OWL SAIL. Layers.

Objects OGM / DCI schema SAIL.

Functional schema SAIL: Dataflow Augmentations, Metadata, Services.

Messages: SAIL Statements IO. Binder / Adapter reactive Message IO: Integrated sources.

Context: Layer / Instance Message shape activation: Quads, Object, Functional SAILs.

Hierarchies (Dataflow): Aggregation matches layer instances specializations. generalizations (layer instances) matches specialisations (layer instances). Aggregated Layers are recursively aggregated.

CDI Layers: SAILs / Rio IO Functional Java APIs Wrappers / Facades for Resources IO (connectors / services / protocols) for Reactive Dataflow Bus sources / processors / sinks. Reactive / Services / Events. SAILs / Rio IO.

CDI Layers: Domains / Contexts Services. Over CDI Layers Functional APIs Wrappers / Facades. Examples: Datasources (Drivers), OGM, OData, etc.

Naming: DIDs Services. [schema.org](http://schema.org), ont.io, [sameAs.org](http://sameas.org). Matching (mappings). WordNet, OpenNLP, Wikipedia. SAIL / Rio.

Reactive (Bus): Vert.x Dataflow: SCDF Pipelines.

Designer: OpenRefine. Protege Web. Editors / browsers (Forms / Flows).

Protocols (SAILs): SPARQL, JSON-LD, JDBC (OGM / OData: Driver DatabaseMetadata). Messages (Functional / Augmentations) Declarative Services (templates).

Clients:

Debug Console.

Declarative UI: ZK / XUL Templates (Content types / Components activation).

Services APIs: Declaratively stated services: templates / queries.

IDE. Runtime Deployment (Application).

Clients: OData / JDBC / WS-\* (CXF) / HATEOAS / etc.

JXTA. DHT. Kafka. Event sourcing. W3C DIDs. Smart Contracts. SoLiD. StratML.

URI: Jersey / CDI APIs. Persistence interface template methods.

(Statement, OldEmployees : Predicate URI / Activation, SalaryUpdate : Mapping URI / Aggregation, Percentage : Function URI / Alignment)

(Domain / Context matching signatures (async streams / topics), Employee : Aggregation / Predicate, Developer : Activation / Mapping / Attributes, ProgrammingLanguages / Alignment / Function / Values);

**To Do:**

* Eclipse.
* RDF4J.
* Core: Deploy / Spring / CDI (JEE). SAILs. Events APIs. Reactive. Vert.x / Kafka / others.
* Core: RDF Quads / Object Mappings / Functional SAILs. Persistence / Representations.
* Matching / Alignments Services: Resolver, Index, Naming, Registry. Lucene. FCA VSM. Events APIs. Reactive.
* Core Integration Services SAILs. Matching: Ontology / Domains / Flows Alignments:
* Apache MetaModel / JBoss Teiid / Services (REST / WS) / Data Sources. Events APIs. Reactive.
* JBoss KIE Services. Events APIs. Reactive.
* Core: Designer. Protege, OpenRefine. Jupyter Notebooks.
* Core: Business Services. OData / JDBC / JCA / OGM / JAF, REST HATEOAS / HAL, WS-\*, DCI / Qi4j. SAILs / Endpoints (Spring / CDI APIs).
* Core: Business Application Backend (Console / Apache Isis).
* Core: Business: Tryton Backend.

**Ontology Matching:**

The typical use case I have in mind is the declaration of equivalence between a flat statement and a property chain, as in the two patterns below:

@prefix : <http://example.org/> .

@prefix ex: <http://ontology.org/example> .

ex:book ex:hasauthor "John".

:book :hasbeencreated :creation\_event .

:creation\_event :carried\_out :person .

:person :is\_identified\_by :appellation .

:appellation rdfs:label "John" .

another important type of equivalence, but slightly different, which I would like to declare is the one between these two patterns:

@prefix : <http://example.org/> .

@prefix ex: <http://ontology.org/example> .

ex:Architect rdfs:label "John" .

:Person :classifiedAs <http://vocab.getty.edu/aat/300024987> ;

rdfs:label "John".

<http://vocab.getty.edu/aat/300024987> a gvp:Concept ;

rdfs:label "architects"@en .

The first one declare an instance of the class Artist according to an ontology(x), the second classify as artist, using a controlled vocabulary term, an instance of a person declared using the ontology (y).

Do you know how can I express such alignments?

I heard about EDOAL, but sincerely I did not fully grasped how to actually use it.

@prefix : <http://example.org/> .

@prefix ex: <http://ontology.org/example> .

ex:book ex:hasauthor "John".

:book :hasbeencreated :creation\_event .

:creation\_event :carried\_out :person .

:person :is\_identified\_by :appellation .

:appellation rdfs:label "John" .

In the scenario above, what is the relation between ex:book and :book, and can you define some rule on how to create one from the other? Is it "same-local-name-but-different-namespace"?

If ex:book and :book are identical then a SHACL sh:equals constraint can be used

ex:BookShape

a sh:NodeShape ;

sh:property [

sh:path ( :hasbeencreated :carried\_out :is\_identified\_by rdfs:label ) ;

sh:equals ex:hasauthor ;

] .

<https://www.w3.org/TR/shacl/#EqualsConstraintComponent>

Note that SHACL includes a syntax for SPARQL-like property path expressions, and the value of sh:path above is a property chain (represented as a simple rdf:List). Other types of paths are supported too: <https://www.w3.org/TR/shacl/#property-paths>

@prefix : <http://example.org/> .

@prefix ex: <http://ontology.org/example> .

ex:Architect rdfs:label "John" .

:Person :classifiedAs <http://vocab.getty.edu/aat/300024987> ;

rdfs:label "John".

<http://vocab.getty.edu/aat/300024987> a gvp:Concept ;

rdfs:label "architects"@en .

I am also unclear whether you want to use alignments to validate constraints (e.g. "does pattern 2 exist for pattern 1"), or to construct/infer one pattern out of the other.

The complexity of the scenario above indicates that you may want to use SPARQL, because then you can more easily look up values through matches, e.g. to match "architects"@en to ex:Architect using some look-up table, and because SPARQL gives you a maximum of expressiveness.

- If you want to validate constraints, you could use SHACL-SPARQL constraints: <https://www.w3.org/TR/shacl/#sparql-constraints>

- If you want to construct target triples, you could use SHACL-AF rules: <https://w3c.github.io/shacl/shacl-af/#rules>

You can write the alignment rules in RIF, then translate them to SPARQL

and use as needed, either as part of a runtime query, or to materialize

your own "equivalence" triples (using SPARQL INSERT or CONSTRUCT).

Rules written in RIF are easier to analyze and document, for example by

expressing them in the RIF XML notation and using XSLT.

I used this approach to align and validate part master and product

structure information in a large dataset derived from different PLM

systems.

What correspondences do you want to express exactly?

From your first example, I understand that you want to express that the

property:

ex:hasauthor

from the first ontology corresponds to the property chain:

:hasbeencreated o :carried\_out o :is\_identified\_by o rdfs:label

in the second ontology.

From your second example, I understand that you want that the class of

ex:Architect in the first ontology correspond to the class of people

that are classified as http://vocab.getty.edu/aat/300024987 .

I wrote these correspondences in an alignment file at:

https://www.emse.fr/~zimmermann/edoal-example.xml

The alignment format from Inria's alignment API is meant to represent

correspondences in a way independent from how the correspondence may be

used. There are different ways of interpreting and using a

correspondence in an alignment:

1. as an ontological axiom

2. as data transformations

3. as schema constraints

4. as "bridge rules" between descriptions of different contexts

Additionally, ontology alignment correspondences may have a "measure"

assigned to them that can be interpreted as a degree of confidence, or

as a fuzzy value, or as a probability, or something else. They also have

additional metadata that makes it clear that they are relating something

from an ontology to something from another ontology. In comparison, a

logical axiom, even if it uses URIs from different namespaces, does not

make this clear.

So, depending on your use case, you may want to use Holger's suggestion

(SHACL) or Paul's (RIF + SPARQL), or something else, but you may also

postpone the decision for later (or leave it to someone else) and just

write an EDOAL alignment like I did. The alignment file can also serve

other purposes, such as alignment evaluation, composition, and enrichment.

**HyTime:**

HyTime (use cases / implementations): XML, XPath, XSL, XLink, XPointer, XQuery use cases. Dimensional Addressing, Rendering (Layers) of Resources representations in contexts. Layers object model: domains to user / services gestures. Declarative templates. Objects DCI / MVC activation (JAF / CDI).

**Scheme:**

I am pleased to announce that we reached the last call for comments on

SRFI-168.

That is, a generic approach to triple and quad store. It support tuples of n-items.

While it might seem overkill to support tuples of n-items, it was somewhat

easy to implement (once you know the math: <https://math.stackexchange.com/q/3146568/23663>). Also, I need such a database

abstraction to implement something (that I think) will be suitable to version

tuples in a direct-acyclic graph (ala git).

This work was inspired from:

"Collaborative Open Data versioning: a pragmatic approach using Linked Data"

Federico Morando, Raimondo Iemma, Simone Basso et Lorenzo Canova

<https://iris.polito.it/handle/11583/2617308>

Hopefully, it will allow to build some kind of portable and scalable wikidata & wikibase.

The very last version of the specification can be read at:

<http://htmlpreview.github.io/?https://github.com/scheme-live/srfi-168/blob/master/srfi-168.html>

A few hints:

- It is supposed to be supported by an ordered key-value (wiredtiger, foundationdb, Oracle BerkeleyDB, TiKV, leveldb, rocksdb...), see SRFI-167 <http://htmlpreview.github.io/?https://github.com/scheme-live/srfi-167/blob/master/srfi-167.html>.

- It can support (only!) scheme base data types but not complex ie. bigint, float, string, symbol, bytevector and vector and list of those preceding. This is implementation dependant but the implementations so far only support those.

- The query language is based on stream-processing. All queries are at least O(n) where n is the length of the seed generator (nstore-from).

- The specification is written in a way that will allow support geospatial indexing

directly in the nstore and outside the nstore but inside the same database so that ACID semantic still holds. More generally, the nstore abstraction can live along side other abstractions in the same okvs database, which allows to implement specific indexing scheme to support geospatial indices or better pagination.

It is still time to make feedback and I hope you do :)

Amirouche ~ amz3

Hello Sebastian,

Glad to read about someone interested in this project.

Le lun. 10 juin 2019 à 17:25, Sebastian Samaruga <ssamarug@gmail.com> a écrit :

Hi,

Sorry for my ignorance but I don't understand "the math".

Me neither, I just explained my problem several times on mathoverflow and at some point, someone came up with an explanation and some else with an algorithm.

Could you please better explain for me what are "prefix permutations"

Good question! The definition is in the stackoverflow question. That is: A is "prefix-permutation" of B, if there is a permutation of A that is prefix of B.

and how they are useful for building indices for SPARQL like queries?

This is useful to me because I rely on an ordered key-value database that is bytes to bytes hashmap where keys are "sorted". Using that database

it is very easy to query for all keys by prefix. Now, when you consider SPARQL queries for a triplestore the WHERE clauses look like the following

(subject, predicate, value)

In my implementation, I consider that ONE or more can be a variable. We can enumerate

those patterns:

(?subject, predicate, object)

(?subject, ?predicate, object)

(?subject, ?predicate, ?object)

(?subject, predicate, ?object)

(subject, ?predicate, ?object)

(subject, predicate, ?object)

(subject, ?predicate, object)

You can recognize that the combinations of (subject, predicate, object) describe all

the patterns, here are those combinations:

('subject',)

('predicate',)

('object',)

('subject', 'predicate')

('subject', 'object')

('predicate', 'object')

So, to be able to bind ANY pattern (in one hop). For all combinations, there must be an index

that can COMPLETE the given combination to form a complete tuple. One can recognize that

the order in which the items of the combination appears in the index doesn't matter. To take the

previous example, both of those indices:

(predicate, object, subject)

And

(object, predicate, subject)

Allow to complete the following pattern:

(?subject, predicate, object)

So the algorithm minimize the set of indices given that prefix-permutation property.

There must be a better explanation.

I've been done some permutations / combinations (rotations) in a model layer of an ongoing project, but still being a newbie and doing much by intuition.

If you dare to skim a very fuzzy and (incoherent?) early drafts, check "Index.docx" at:

https://github.com/snxama/scrapbook

I just skimmed over it. It very fuzzy. I think it would help if you had a table of content and write down the introduction and the conclusion of your work beforehand.

I don't know if this is feedback for what you are doing. I'm trying to build core ontologies for semantic / object mappings for declarative modelling of "business domains" inferred from aggregating and blending distributed sources of data.

aggregating different sources of data is difficult. I will not forbid you from trying.

As I'm in a stage of choosing a platform for the development of some kind of prototype, I'll be choosing an n-store implementation.

I would be very glad if you implemented my nstore in Java. Wiredtiger has bindings in Java. There might other bindings or ordered key-value store available in Java. FoundationDB is available in Java.

Maybe, I think, yours seems very "flexible" although I haven't reviewing very much other alternatives yet.

That work started as a learning project. Like I said in the original mail, I stumble upon an interesting problem (how to version triples) and thought that it would be fun to solve.

Now, it happens that for my (want-to-be) natural language processing (want-to-be) research, I will use the nstore.

Feel free to ask more question. Also I am following you on github.

Sure!

You can watch this repository, eventually, that is where I will put everything https://github.com/awesome-data-distribution/datae

Le lun. 10 juin 2019 à 20:43, Sebastian Samaruga <ssamarug@gmail.com> a écrit :

Hello,

Thanks a lot for your response. I'd like to hear more about your project. For Java, there is Clojure, a LISP dialect that runs natively and integrates seamlessly with all Java ecosystem, frameworks and other VM languages.

Maybe I'm so dumb but, could you define the predicate isPrefixOf(statement1, statement2).

"abc" is prefix of "abcdef".

What makes a tuple being prefix of another?

It is not a tuple that is prefix of another. It is the combination (or a permutation of it) that is prefix of the tuple representing the index.

For instance, the following combination:

(subject, predicate)

is prefix of:

(subject, predicate, object)

(object) is prefix of:

(object, subject, predicate)

Also (subject, object) is prefix of

(subject, object, predicate)

and (subject, object) permutation-prefix of:

(object, subject, predicate)

because there is a permutation that is (object,subject) that is prefix of it.

Hope it helps!

I created a repository to make it easier to test a real implementation

of that specification: <https://github.com/scheme-live/srfi-167-and-168-tutorial#setup>

On preliminary benchmarks that code is faster than blazegraph!

This is not perfect (one of the cloned repository is 1G big) but it should get you going.

Also it does require a linux box but doesn't require system installation.

I made progress regarding the functional store aka. fstore ie. a versioned generic

tuple store (which is the reason I needed the nstore and eventually discovered it

can be useful in other contexts).

It is bundled in the above repository, it is not documented [1] but there is a beginning of test

suite at:

https://git.sr.ht/~amz3/chez-scheme-arew/tree/master/src/arew/data/base/fstore-test.scm

[1] well, the API of fstore is the same as the API nstore, except you have to specify a branch

against which the query must happen.

Le sam. 8 juin 2019 à 13:51, Amirouche Boubekki <amirouche.boubekki@gmail.com> a écrit :

I am pleased to announce that we reached the last call for comments on

SRFI-168.

That is, a generic approach to triple and quad store. It support tuples of n-items.

While it might seem overkill to support tuples of n-items, it was somewhat

easy to implement (once you know the math). Also, I need such a database

abstraction to implement something (that I think) will be suitable to version

tuples in a direct-acyclic graph (ala git).

I got a question off-list about what permutation-prefix means and where it is

useful.

There is a sister question on stackoverflow that try to explain in terms of programming what problem I am trying to solve: <https://stackoverflow.com/q/55143485/140837>

I will work on a document that will better explain how it is used and why

it is useful.

I am very pleased to announce the immediate availability of nomunofu.

nomunofu is database server written in GNU Guile that is powered by WiredTiger ordered key-value store.

It allows to store and query triples. The goal is to make it much easier, definitely faster to query as big as possible tuples of three items. To achieve that goal, the server part of the database is made very simple and it only knows how to do pattern matching. Also, it is possible to swap the storage engine to something that is horizontally scalable and resilient.

The client must be smarter, and do as they please to full-fill user requests. Today release only include a minimal Python client. In the future, I plan to extend the Python client to fully support SPARQL 1.1.

Preliminary tests over 100 000 and 1 000 000 triples are good looking. Next step is to reach 1 billion triples and eventually 9 billions wikidata triples.

You can get the code with the following command:

git clone https://github.com/amirouche/nomunofu

After the installation of GNU Guix [0], you can do:

make init && gunzip test.nt.gz && make index && make web

And in another terminal:

make query

Thanks to Guix, portable binaries for amd64 Ubuntu 18.04 will be made available in a few weeks, along with this, a docker image will be built. The binary release will include wikidata pre-loaded.

[0] https://guix.gnu.org/download/

Here is an example ipython session:

$ ipython

Python 3.7.3 (default, Oct 7 2019, 12:56:13)

Type 'copyright', 'credits' or 'license' for more information

IPython 7.10.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from nomunofu import Nomunofu

In [2]: from nomunofu import var

In [3]: nomunofu = Nomunofu('http://localhost:8080');

In [4]: nomunofu.query((var('uid'), "http://www.w3.org/2000/01/rdf-schema#label", "Belgium"))

Out[4]: [{'uid': 'http://www.wikidata.org/entity/Q31'}]

In [5]: nomunofu.query((var('uid'), "http://www.w3.org/2000/01/rdf-schema#label", "Belgium"), (var('about'), "http://

...: schema.org/about", var('uid')))

Out[5]:

[{'uid': 'http://www.wikidata.org/entity/Q31',

'about': 'https://www.wikidata.org/wiki/Special:EntityData/Q31'},

{'uid': 'http://www.wikidata.org/entity/Q31',

'about': 'https://it.wikivoyage.org/wiki/Belgio'},

{'uid': 'http://www.wikidata.org/entity/Q31',

'about': 'https://an.wikipedia.org/wiki/Belchica'},

{'uid': 'http://www.wikidata.org/entity/Q31',

'about': 'https://sl.wikipedia.org/wiki/Belgija'},

{'uid': 'http://www.wikidata.org/entity/Q31',

'about': 'https://pfl.wikipedia.org/wiki/Belgien'},

{'uid': 'http://www.wikidata.org/entity/Q31',

'about': 'https://crh.wikipedia.org/wiki/Bel%C3%A7ika'},

{'uid': 'http://www.wikidata.org/entity/Q31',

'about': 'https://fiu-vro.wikipedia.org/wiki/Belgi%C3%A4'},

{'uid': 'http://www.wikidata.org/entity/Q31',

'about': 'https://fr.wikipedia.org/wiki/Belgique'}

...

Cheers,

Amirouche ~ zig ~ <https://hyper.dev>

Le dim. 8 déc. 2019 à 18:52, Amirouche Boubekki

<amirouche.boubekki@gmail.com> a écrit :

>

> I am very pleased to announce the immediate availability of nomunofu.

>

> nomunofu is database server written in GNU Guile that is powered by WiredTiger ordered key-value store.

>

> It allows to store and query triples. The goal is to make it much easier, definitely faster to query as big as possible tuples of three items. To achieve that goal, the server part of the database is made very simple and it only knows how to do pattern matching. Also, it is possible to swap the storage engine to something that is horizontally scalable and resilient.

>

I pushed portable binaries built with gnu guix for amd64 with a small

database file. You can download it with the following command:

$ wget https://hyper.dev/nomunofu-v0.1.3.tar.gz

The uncompressed directory is 7GB.

Once you have downloaded the tarball, you can do the following cli

dance to run the database server:

$ tar xf nomunofu-v0.1.3.tar.gz && cd nomunofu && ./nomunofu serve 8080

The database will be available on port 8080. Then you can use the

python client to do queries.

Here is example run on the current dataset, that queries for instance

of (P31) government (Q3624078):

In [1]: from nomunofu import Nomunofu

In [2]: from nomunofu import var

In [3]: nomunofu = Nomunofu('http://localhost:8080');

In [4]: nomunofu.query((var('uid'),

'http://www.wikidata.org/prop/direct/P31',

'http://www.wikidata.org/entity/Q3624078'), (var('uid'),

'http://www.w3.org/2000/01/rdf-schema#label', var('label')))

Out[4]:

[{'uid': 'http://www.wikidata.org/entity/Q31', 'label': 'Belgium'},

{'uid': 'http://www.wikidata.org/entity/Q183', 'label': 'Germany'},

{'uid': 'http://www.wikidata.org/entity/Q148', 'label': 'China'},

{'uid': 'http://www.wikidata.org/entity/Q148',

'label': "People's Republic of China"},

{'uid': 'http://www.wikidata.org/entity/Q801', 'label': 'Israel'},

{'uid': 'http://www.wikidata.org/entity/Q45', 'label': 'Portugal'},

{'uid': 'http://www.wikidata.org/entity/Q155', 'label': 'Brazil'},

{'uid': 'http://www.wikidata.org/entity/Q916', 'label': 'Angola'},

{'uid': 'http://www.wikidata.org/entity/Q233', 'label': 'Malta'},

{'uid': 'http://www.wikidata.org/entity/Q878',

'label': 'United Arab Emirates'},

{'uid': 'http://www.wikidata.org/entity/Q686', 'label': 'Vanuatu'},

{'uid': 'http://www.wikidata.org/entity/Q869', 'label': 'Thailand'},

{'uid': 'http://www.wikidata.org/entity/Q863', 'label': 'Tajikistan'},

{'uid': 'http://www.wikidata.org/entity/Q1049', 'label': 'Sudan'},

{'uid': 'http://www.wikidata.org/entity/Q1044', 'label': 'Sierra Leone'},

{'uid': 'http://www.wikidata.org/entity/Q912', 'label': 'Mali'},

{'uid': 'http://www.wikidata.org/entity/Q819', 'label': 'Laos'},

{'uid': 'http://www.wikidata.org/entity/Q298', 'label': 'Chile'},

{'uid': 'http://www.wikidata.org/entity/Q398', 'label': 'Bahrain'},

{'uid': 'http://www.wikidata.org/entity/Q12560', 'label': 'Ottoman Empire'}]

As of right now there is less than 10 000 000 triples that were

imported. Blank nodes are included, but only english labels are

imported.

You can find the source code at:

https://github.com/amirouche/nomunofu

I hope you have a good day!

I am pleased to share with you the v0.1.4 binary release. It contains

the following improvements:

- The REST API takes JSON as input, which will make it easier to

create clients in other programming languages;

- The REST API takes limit and offset as query string. The maximum

limit is 1000;

- There is better error handling, the server will return a HTTP status

code 400 if it detects an error;

- Add aggregation queries `sum`, `count` and `average`, see the Python

client (nomunofu.py) to know how to properly format the query;

- Python client method `Nomunofu.query(\*patterns, limit=None,

offset=None)` returns a generator.

Also, the harmless warnings are silenced. The database files are

compatible with the previous release.

This release comes with full wikidata lexemes triples.

You can download the amd64 portable binary release plus database files

with the following command:

wget http://hyper.dev/nomunofu-v0.1.4.tar.bz2

The directory is 11G uncompressed.

Grab the source code with the following command:

git clone https://github.com/amirouche/nomunofu

Here is an example Python query that returns at most 5 adverbs:

In [10]: for item in nomunofu.query(

...: (var('uid'), wikibase('lexicalCategory'),

'http://www.wikidata.org/entity/Q380057'),

...: (var('uid'), rdfschema('label'), var('label')),

...: limit=5):

...: print(item)

...:

{'uid': 'http://www.wikidata.org/entity/L3244', 'label': 'always'}

{'uid': 'http://www.wikidata.org/entity/L4124', 'label': 'here'}

{'uid': 'http://www.wikidata.org/entity/L4326', 'label': 'often'}

{'uid': 'http://www.wikidata.org/entity/L5201', 'label': 'too'}

{'uid': 'http://www.wikidata.org/entity/L5321', 'label': 'yet'}

Cheers,

Hello all ;-)

I ported the code to Chez Scheme to do an apple-to-apple comparison

between GNU Guile and Chez and took the time to launch a few queries

against Virtuoso available in Ubuntu 18.04 (LTS).

Spoiler: the new code is always faster.

The hard disk is SATA, and the CPU is dubbed: Intel(R) Xeon(R) CPU

E3-1220 V2 @ 3.10GHz

I imported latest-lexeme.nt (6GB) using guile-nomunofu, chez-nomunofu

and Virtuoso:

- Chez takes 40 minutes to import 6GB

- Chez is 3 to 5 times faster than Guile

- Chez is 11% faster than Virtuoso

Regarding query time, Chez is still faster than Virtuoso with or

without cache. The query I am testing is the following:

SELECT ?s ?p ?o

FROM <http://fu>

WHERE {

?s <http://purl.org/dc/terms/language> <http://www.wikidata.org/entity/Q150> .

?s <http://wikiba.se/ontology#lexicalCategory>

<http://www.wikidata.org/entity/Q1084> .

?s <http://www.w3.org/2000/01/rdf-schema#label> ?o

};

Virtuoso first query takes: 1295 msec.

The second query takes: 331 msec.

Then it stabilize around: 200 msec.

chez nomunofu takes around 200ms without cache.

There is still an optimization I can do to speed up nomunofu a little.

Happy hacking!

<https://github.com/amirouche/copernic>

<http://copernic.space/>

Scheme tuple store: Rio / SAIL (Form / Flows Cons Zipper Shapes).

**Zippers:**

Zippers: car / cdr on tree / list structures with predicates / iteration / recursion (reference model, contexts / occurrences). Shape Monads. E.g.: Uncle (reified relation predicate and reified Relation w./ roles / attributes). Dynamic Shape Monad on Kinds. Aggregation, Activation, Alignments.

Example: Marriage (TBD).

Predicates:

:aHusband :marriedWith :aWife

:marriedWith rdfs:domain :Male

:marriedWith rdfs:range :Female

Relationship:

(aMarriage : Relation, anStatement : marriageStatement, aKind : husbandRole, aResource : aHusband);

(aMarriage : Relation, anStatement : marriageStatement, aKind : wifeRole, aResource : aWife);

(Marriage : Relationship, Marriages : Relation, anStatement : marriagesStatements, aKind : marriageRole);

Predicates / Relationships, Relationships / Predicates entailment. Dimensional: inference / relation types / restrictions.

Encode order / hierarchies / relations (parent / child, prev / next, etc.) / iterations / conditionals / jumps.

Dimensional Domain: dimensions, units, measures, values. Comparisons, relations. State. Events (marriage example). Verbs (action, passion, state). Order (data / schema / behavior).

**Proposal for representing Aggregate Statistical Data (Guha):**

This document can be accessed here:

<https://docs.google.com/document/d/139jXakeQk4ChwCkGjqq5wJfCPMDnwIV94oCH-JzJrhM/edit?usp=sharing>

Look forward to feedback.

Guha

Representing aggregate statistics

Examples of aggregate statistical reports include those from Census Organizations (e.g., American Community Survey), Health Organizations (e.g., CDC Wonder) and many others. This is a schema, currently in use on DataCommons.org for representing facts stated in these reports. This document describes certain general mechanisms for representing statistical populations and associated observations. This document will be followed later by a companion proposal suggesting some basic common vocabulary useful for representing the kind of data released by the US Census, CDC, etc.

Our interest is not in describing a data set or mapping columns in csv files, but in representing the actual data itself. Other efforts have focused on characterizing data cubes in terms of dimensions, etc. While we draw upon their work, our goals are different.

Examples of the kind of statistics we would like to represent include:

1. In 2016, there were 1213 people in East Podunk, California, who were male, married, with a median age of 22.

2. In 2017, there were 20 deaths in Falooda County where the cause of death was XYZ

We will refer to ‘number of people who are male, hispanic’, ‘number of deaths where cause of death was XYZ’, etc. as variables. Since the number of possible variables increases combinatorially, clearly, we can’t have a properties for each variable (or worse, property for each variable x years). We need a way of compositional way of constructing variable references. We use the concept of a StatisticalPopulation to do this construction.

A StatisticalPopulation is a set of instances of a certain given type that satisfy some set of constraints. The property populationType is used specify the type. Any property that can be used on instances of that type can appear on the statistical population. An instance of StatisticalPopulation whose populationType is C1, which has the properties p1, p2, … with values v1, v2, … corresponds to the set of objects of type C1 what have the property p1 with value v1, property p2 with value v2, etc. The properties numConstraints and constrainingProperties are used to specify which of the populations properties are used to specify the population. In the two examples above:

Node: SP1

type: StatisticalPopulation

populationType: Person

location: EastPodunkCalifornia

gender: Male

maritalStatus: Married

numConstraints: 3

constrainingProperties: location, gender, race

Node: SP2

type: StatisticalPopulation

populationType: MortalityEvent

location: FaloodaCounty

causeOfDeath: XYZ

numConstraints: 2

constrainingProperties: location, causeOfDeath

SP1 is an abstract set in the sense that it does not correspond to a particular set of people who satisfy that constraint at a certain point in time, but rather, to an abstract specification, about which we can make observations that are grounded at a particular point in time. We now turn our attention to the representation of these observations.

Instances of the class Observation are used to specify observations about an entity (which may or may not be an instance of a StatisticalPopulation), at a particular time. The principal properties of an Observation are observedNode, measuredProperty, measuredValue (or median, etc.) and observationDate (measuredProperty can, but need not always, be w3c rdf data cube "measure properties", as in lifeExpectancy example here: https://www.w3.org/TR/vocab-data-cube/#dsd-example.) In the two examples above:

Node: Obs1

type: Observation

observedNode: SP1

measuredProperty: age

median: “23 years”

observationDate: “2016”

Node: Obs2

type: Observation

observedNode: SP1

measuredProperty: count

measuredValue: 1213

observationDate: “2016”

Node: Obs3

type: Observation

observedNode: SP2

measuredProperty: count

measuredValue: 20

observationDate: “2017”

Observations can also have properties related to the measurement technique, margin of error, etc. To elaborate on Obs2 above, we can have:

Node: Obs2

type: Observation

observedNode: SP1

measuredProperty: count

measuredValue: 1213

observationDate: “2016”

marginOfError: 22

measurementMethod: CensusACS5yrSurvey

Notes:

1. Care needs to be exercised when querying StatisticalPopulations, to make sure that the query specifies all the constraining properties.

2. We do not yet have a way of using properties which are named in the opposite direction e.g. we handle "alumniOf" (relating a person to an org), but if the only existing property was "alumni" (relating an org to a person).

**HL7**

HL7 v3 can not be fully implemented with the current state of Object Databases.

I'm interested in HL7 and in particular FHIR.

FHIR has an RDF/JSONLD representation, and the schema is represented as SHEX (amongst several other implementations, including JSON/JSON schema, XML/RelaxNG etc).

I think this is a very interesting piece of work and I'm interested in collaborations.

I am participating in the European H2020 project named GATEKEEPER which is focused on smart personal healthcare. HL7 is a member of the project. We’re looking at combining the Web of Things, FHIR, semantic technologies and more in a variety of pilots. FHIR is essential a means to use RESTful interfaces to access healthcare related data in a variety of formats, e.g. JSON-LD. There is a body of work on healthcare related ontologies. Here is a pointer to FHIR by the UK’s NHS, and the overview from FHIR itself:

<https://digital.nhs.uk/services/fhir-apis>

<https://www.hl7.org/fhir/overview.html>

A major challenge is how to integrate heterogeneous data sources whilst avoiding a loss of valuable information due to differences in vocabularies, data quality, context and so forth.

**StratML:**

Can AI be used on StratML utilizing ontologies to create customizable dashboards for project management and collaboration for large networks of collaborating people from different fields of work?

Question is of importance in eGovernment, eGovernance and achieving sustainable development goals.

XML constitutes a structure for language, and certain kinds of AI can be built

using structured language expressions.

According to Wikipedia, intelligence involves perceiving the environment and acting to maximize the chance of achieving goals. https://en.wikipedia.org/wiki/Artificial\_intelligence The article also cites Tesler's Theorem: "AI is whatever hasn't been done yet." See also https://en.wikipedia.org/wiki/AI\_effect

While it may be possible for intelligent agents to decipher goals from unstructured text, it seems likely they might be able to more effectively help us achieve our goals if we make them explicit in terms of near-term objectives and performance indicators. Since that has not yet been done on a worldwide scale, why should it not begin with us?

There are more than 4,000 plans in the StratML collection that can be used for demonstration purposes, including the SDGs: <https://stratml.us/drybridge/index.htm#UNSDG>

As far as I am aware, however, progress against the SDGs is not being reported in an open, standard, machine-readable format. See, for example, the HTML, CSV & PDF at https://unstats.un.org/sdgs/report/2019/ The presentation of the data is actually pretty good. See, for example, https://unstats.un.org/sdgs/report/2019/goal-17/

The problem is that few people know about and take the time to view it, much less to do anything about it. While making the data available in open, standard, machine-readable format would not solve that problem, it would make it easier for value-added intermediaries to engage stakeholders in ways that are more accessible and meaningful to them, e.g., in their own personal and organizational/corporate performance plans. From my perspective, failing to do so is an example of artificial ignorance. https://www.linkedin.com/pulse/artificial-ignorance-owen-ambur/

Here's an article addressing the role of AI in achieving the SDGs: https://www.nature.com/articles/s41467-019-14108-y It is relatively long and unstructured and I haven't taken the time to read all of it. However, here some key point:

We therefore recommend that AI applications that target SDGs are open and explicit about guiding ethical principles, also by indicating explicitly how they align with the existing guidelines.

See StratML tool, app, and service requirements Objective 8.3: Values Alignment & Goal 9: Values Validation. Those requirements are prime candidates for the application of AI.

On the other hand, the lack of interpretability of AI, which is currently one of the challenges of AI research, adds an additional complication to the enforcement of such regulatory actions ...

AI developers and agents should be expected, if not required, to document their objectives and performance indicators in an open, standard format that is both human- and machine-readable.

... associations such as the Future of Life Institute are reviewing and collecting policy actions and shared principles around the world to monitor progress towards sustainable-development-friendly AI ...

Perhaps we should schedule a televideo conference for them to brief us on their AI activities and explore prospects for collaboration. <https://stratml.us/carmel/iso/FLIwStyle.xml>

A global and science-driven debate to develop shared principles and legislation among nations and cultures is necessary to shape a future in which AI positively contributes to the achievement of all the SDGs.

Whenever I see calls for legislation and regulation, I view it both as a cop out as well as an attempt to dictate to others that which we should take upon ourselves to do, in partnership with those who share our values and objectives. Perhaps those who view it as the best way forward should either study the China model ... or just stand idly by and wait for them to impose it on all of us. For an alternative model, see https://www.linkedin.com/pulse/privately-well-practiced-public-policymaking-owen-ambur/

Here's a shorter article on AI & the SDGs: https://www.undp.org/content/undp/en/home/blog/2019/Using\_AI\_to\_help\_achieve\_Sustainable\_Development\_Goals.html It suggests, "To improve data accessibility, for example, collectors and generators of data, whether governments or companies, will need to grant greater access to NGOs and others seeking to use the data for public service." However, it shows no awareness of the importance of open, machine-readable data standards -- perhaps because usage of such standards would reduce the need for consultants ... or, rather, it would require them to add higher-level values than massaging amorphous, aimless data.

With respect to dashboards, see https://gcn.com/articles/2017/09/12/yet-another-dashboard.aspx & https://gcn.com/articles/2017/05/10/machine-readable-data.aspx

I look forward to learning what we might be able to do together along these lines.

**Monads:**

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)

Monad: wrapper type. Metaclass.

Monad: wrapped type. Class.

Monad: wrapped value. Instance.

Monad: wrappers hierarchy context type instance. Occurrence (mapping results).

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.

Data: product price / marital status.

Information: price variation / state change.

Knowledge: increase, decrease / marriage, divorce.

TBD.

**Resources:**

<https://dzone.com/articles/understanding-inversion-of-control-and-dependency?edition=586296&utm_source=Zone+Newsletter&utm_medium=email&utm_campaign=java+2020-03-23>

<https://dzone.com/articles/getting-started-with-ontotext-graphdb-and-rdf4j>

<https://medium.com/wolox/reactor-java-meets-reactive-programming-16105c026fc3>

<https://spring.io/blog/2016/06/07/notes-on-reactive-programming-part-i-the-reactive-landscape>

<https://spring.io/reactive>

<https://spring.io/event-driven>

<https://github.com/CarbonLDP/rdf4j-spring>

<https://www.ontotext.com/>

<https://dzone.com/articles/dci-architecture-is-visionary?utm%5C_medium=feed&utm%5C_source=feedpress.me&utm%5C_campaign=Feed%5C:%20dzone%5C%2Fjava>

<https://dzone.com/articles/implementing-dci-qi4j>

<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://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>

<https://www.baeldung.com/java-spi>

<https://itnext.io/java-service-provider-interface-understanding-it-via-code-30e1dd45a091>

<https://www.oracle.com/technical-resources/articles/middleware/luttikhuizen-adapters.html>

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

<https://www.w3.org/community/graphql-rdf/>

<https://json-ld.org/>

<https://comunica.github.io/Article-ISWC2018-Demo-GraphQlLD/>

<https://en.m.wikipedia.org/wiki/SHACL>

<https://shex.io/>

<https://en.m.wikipedia.org/wiki/SPARQL>

<https://www.categoricaldata.net/>

<http://logicprogramming.stanford.edu/public/index.php>

<https://markhneedham.com/blog/2020/02/04/neo4j-enriching-existing-graph-wikidata-sparql-api>

<https://www.depositonce.tu-berlin.de/bitstream/11303/2617/2/Dokument_29.pdf>

<https://www.cambridge.org/core/books/introduction-to-coalgebra/0D508876D20D95E17871320EADC185C6#>

<https://www.slideshare.net/mobile/PatHayes/blogic-iswc-2009-invited-talk>

<https://medium.com/@bblfish/epistemology-in-the-cloud-472fad4c8282>

<https://academic.oup.com/comjnl/article-abstract/54/1/31/336864>

<https://math.stackexchange.com/questions/2896172/how-should-one-model-rdf-semantics-in-terms-of-category-theory>

<https://www.w3.org/TR/2012/REC-owl2-mapping-to-rdf-20121211/>

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.159.2020&rep=rep1&type=pdf>

<https://bartoszmilewski.com/>

<https://www.youtube.com/user/DrBartosz/playlists>

<http://www.tac.mta.ca/tac/reprints/articles/22/tr22.pdf>

<https://math.stackexchange.com/questions/2896172/how-should-one-model-rdf-semantics-in-terms-of-category-theory>

<http://arxiv.org/pdf/1109.0983v1>

<http://iws.cs.uni-magdeburg.de/~mossakow/>

<http://www.aifb.kit.edu/web/Techreport893>

<http://journal.ub.tu-berlin.de/eceasst/article/view/158/142>

<http://wwwhome.ewi.utwente.nl/~fokkinga/mmf92b.pdf>

<http://www.staff.science.uu.nl/~ooste110/syllabi/catsmoeder.pdf>

<http://bblfish.net/>

<https://www.slideshare.net/PatHayes/rdf-with-contexts>

<http://ceur-ws.org/Vol-166/70.pdf>

<http://www-formal.stanford.edu/guha/>

<http://dig.csail.mit.edu/2006/Papers/TPLP/n3logic-tplp.pdf>

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.159.2020&rep=rep1&type=pdf>

<https://pdfs.semanticscholar.org/40bb/e9978e2c4080740f55634ac58033bfb37d36.pdf>

<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.40.7008>

<https://skillsmatter.com/skillscasts/10179-the-maths-behind-types>

<https://math.stackexchange.com/questions/2905226/what-kind-of-categorical-object-is-an-rdf-model/2905287#2905287>

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.98.4812&rep=rep1&type=pdf>

<https://pdfs.semanticscholar.org/4899/493c11d2e803bb86ef6b849fb7b3185be1e3.pdf>

<https://eprints.soton.ac.uk/270829/>

<https://math.stackexchange.com/questions/2921477/what-does-the-category-of-rdf-models-look-like-in-institution-theory>

<https://www.sciencedirect.com/science/article/pii/S0304397502003663>

<https://www.wisnesky.net/dbpl15.pdf>

<https://homes.cs.washington.edu/~chushumo/files/cosette_pldi17.pdf>

<http://categoricaldata.net/>

<https://pdfs.semanticscholar.org/b8c8/5a3e7a04020259ec9a58c7e5563033f52844.pdf>

<https://arxiv.org/abs/1907.00855>

<https://arxiv.org/abs/1902.00545>

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.481.7519&rep=rep1&type=pdf>

<https://www.iep.utm.edu/insti-th/>

<https://www.categoricaldata.net/papers>

<https://www.wisnesky.net/>

<https://twitter.com/bblfish/status/1165195822625153024>

<https://skillsmatter.com/skillscasts/10179-the-maths-behind-types>

<https://twitter.com/hmemcpy/status/1160870623943561216>

<https://www.youtube.com/user/DrBartosz/playlists>

<https://typelevel.org/cats/>

<https://www.sciencedirect.com/science/article/pii/S0890540112001010>

<https://arxiv.org/abs/1902.00545>

<http://math.mit.edu/~dspivak/CT4S.pdf>

<https://arxiv.org/abs/1009.1166>

<https://arxiv.org/pdf/1706.00526.pdf>

<https://www.sciencedirect.com/science/article/pii/S1571066107000746>

<https://cstheory.stackexchange.com/questions/42689/what-logic-correponds-via-curry-howard-to-a-monad>

<https://arxiv.org/abs/1907.00855>

<https://arxiv.org/abs/1902.00545>

<https://www.categoricaldata.net/help/index.html>

<https://www.categoricaldata.net/help/FOAF.html>

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.119.5368&rep=rep1&type=pdf>

<https://eprints.soton.ac.uk/270829/>

<https://web-cats.gitlab.io/>

<http://shex.io/extensions/Map/>

<https://github.com/hmemcpy/milewski-ctfp-pdf>

<http://people.cs.ksu.edu/~hitzler/resources/publications/fois06.pdf>

<http://people.cs.ksu.edu/~hitzler/resources/publications/cando05.pdf> (pushouts)

<https://www.w3.org/TR/shacl/#EqualsConstraintComponent>

<https://www.w3.org/TR/shacl/#property-paths>

<https://www.w3.org/TR/shacl/#sparql-constraints> (construct target triples).

<https://w3c.github.io/shacl/shacl-af/#rules>

<https://www.emse.fr/~zimmermann/edoal-example.xml>

<https://twitter.com/bblfish/status/1218559825476562945>

<https://twitter.com/bblfish/status/1215024256985247745>

<https://www.slideshare.net/PatHayes/blogic-iswc-2009-invited-talk>

<https://ruben.verborgh.org/publications/verborgh_phd_2014/>

<https://www.brics.dk/LS/98/2/BRICS-LS-98-2.pdf>

<https://csarven.ca/linked-research-decentralised-web>

<https://dzone.com/articles/implementing-dci-qi4j>