Title

Antoine Gréa

Abstract

Acknowledgements

Table of Content

Introduction

0.1 Thesis Context

0.2 Motivations

0.2.1 Observation

0.2.2 Abstraction

0.2.3 Cognition

0.3 Issues

0.4 Contributions

0.5 Plan

1 Knowledge representation

Knowledge representation is at the intersection of maths, logic, language and computer sciences. Its research begins in the end of the 19th century, with Cantor inventing set theory (Cantor 1874). Then after a crisis in the begining of the 20th century with Russel's paradox and Gödel's incompletude theorem, revised versions of the set theory become one of the fundations of mathematics. The most accepted version is the Zermelo–Fraenkel axiomatic set theory with the axiom of Choice (ZFC) (Fraenkel *et al.* 1973, vol. 67; Ciesielski 1997). This effort was to represent mathematics in itself as a way to formalize its own expression.

A similar process happened in the 1970's, when logic based knowledge representation gained popularity among computer scientists (Baader 2003). Systems at the time explored notions such as rules and networks to try and organize knowledge into a rigourous structure. At the same time other systems were built arround First Order Logic (FOL). Then, around the 1990's, the research began to merge in search of common semantics in what led to the developpement of Description Logics (DL). This domain is expressing knowledge as a hierarchy of classes containing individuals.

From there and with the advent of the world wide web, engineers were on the lookout for standardization and interoperability of computer systems. One such standardization took the name of "semantic web" and aimed to created a widespread network of connected services sharing knowledge between one another in a common language. At the begining of the 21st century, several languages were created, all based around W3C specifications called Resource Description Framework (RDF) (Klyne and Carroll 2004). This language is based around the notion of statements as triples. Each can store a unit of knowledge at a time. All the underlying theoritical work of DL continued with it and created more efficient and lighter derivatives. One such derivative is the family of languages called Web Ontology Language (OWL) (Horrocks *et al.* 2003). Ontologies and knowledge graphs are more recent names for the representation and definition of categories (DL classes), properties and relation between concepts, data and entities.

All these knowledge description systems rely on a syntax to interoperate systems and users to one another. The base of such language comes from the formalization of automated grammars by Chomsky (1956). It mostly consist in a set of hierarchical rules aiming to deconstruct an input string into a sequence of terminal symbols. This deconstruction is called parsing and is a common operation in computer science. More tools for the formalization of computer language emerged soon after thanks to Backus (1959) while working on a programming language at IBM. This is how the Backus–Naur Form (BNF) meta-language was created ontop of Chomsky's formalization.

All these tools are the base for all modern knowledge representations. In the rest of this chapter, we discuss the fundamentals of each of the aspects of knowledge description, then we propose a knowledge description framework that is able to adapt to its usage.

1.1 Fundamentals

First and foremost, we present the list of notations in this document. While trying to stick to traditional notations, we also aim for an unambiguious notation across several domains while remaining concise and precise. In table 1.1 we present the list of all symbols and relations. When possible we use the classical mathematical operator and otherwise we prefer lower case greek letters for relations. Table 1.2 lists all sets and general notions that we represent with uppercase letters.

1.1.1 Fundation of maths and logic systems

In order to understand knowledge representation, some mathematical and logical tools need to be presented.

1.1.1.1 Logic

The first mathematical notion we define is logic. More precisely First Order Logic (FOL) in the context of DL.

Introducing the two boolean values $\top$ *true* and $\bot$ *false* along with the classical boolean operators $\neg$ *not*, $\land$ *and* and $\lor$ *or*. These are defined the following way :

- $\neg\top = \bot$
- $a \land b \vDash (a = b = \top)$
- $\neg(a \lor b) \vDash (a = b = \bot)$

With $a \vDash b$ being the logical implication also called entailement.

Table 1.1: List of symbols and relations. The symbol $\pm$ shows when the notation has signed variants.

Symbol	Description
$var : exp$	The colon is a separator to be read as "such that".
$[exp]$	Iverson's brackets: $[false] = 0 \wedge [true] = 1$.
$\{e : exp(e)\}$	Set builder notation, all e such that $exp(e)$ is true.
$\langle e_1, e_2, e_n \rangle$	n-Tuple of various elements.
$e_s \xrightarrow{e_p} e_o$	Link or edge from e_s to e_o having e_p.
$l \downarrow a$	Link l partially supports step a.
$\pi \Downarrow a$	Plan π fully supports a.
$\varnothing$	Empty set, also notted $\{\}$.
$\subset, \cup, \cap, \setminus$	Set inclusion, union, intersection and difference.
$e \in E$	Element e belongs in set E also called choice operator.
$=, \neq$	Equal and not equal.
$\neg, \wedge, \vee$	Negation (not), conjonction (and), disjonction (logical or).
$\|E\|$	Cardinal of set E.
$t_1 \prec t_2$	t_1 subsumes t_2. Also used for order : t_1 precedes t_2
$\top, \bot$	Top and bottom symbols used as true and false respectively.
$\models$	Entails, used for logical implication.
$f \circ g$	Function composition.
$\otimes$	Flaws in a partial plan. Several variants :
$\otimes_{\ddagger}$	• unsupported subgoal.
$\otimes_{\dagger}$	• causal threat to an existing causal link.
$\otimes_{\circlearrowleft}$	• cycle in the plan.
$\otimes_{*}$	• decomposition of a compound action.
$\otimes_{\multimap}$	• alternative to an existing action.
$\otimes_{\circledcirc}$	• orphan action in the plan.
$\oplus, \ominus$	Positive and negative resolvers.
¢	Cost of an action.
$\mathbb{i}, \mathbb{g}$	Initial and Goal step.
$\mathbb{P}(E)$	Powerset, set of all subsets of E.
$\mathcal{P}(X)$	Probability of event X.
pre, *eff*	Preconditions and effects of an action.
δ	Duration of an action.
$\mu^{\pm}(e)$	Signed meta-relation. μ^+ abstracts and μ^- reifies.
	μ alone gives the abstraction level.
$\nu(e)$	Name of entity e.
π	Plan or method of compound action.
$\rho(e)$	Parameters of entity e.
$\sigma(e)$	Scope of entity e.
$\tau(e)$	Type of entity e.
$\phi^{\pm}(e)$	Signed incidence (edge) and adjacence (vertice) function for graphs.
	ϕ^- gives the anterior (subject), ϕ^+ the posterior (object) and
	ϕ^0 gives the property of statement or cause of a causal link.
	ϕ alone gives a triple corresponding to the edge.
$\chi^*(g)$	Transitive cover of graph g.

Table 1.2: List of important named sets.

Name	Description
A	Actions.
C	Causal links.
D	Domain of knowledge.
E	Entities.
$\mathcal{E}$	Edges of a graph.
F	Fluents.
L	Literals.
O	Observations.
P	Properties.
Q	Quantifiers.
S	Statements.
T	Types.
V	Vertex of a graph.
Π	Plans or method of compound action.

1.1.1.2 Set theory

We also need to define the notion of set. At the begining of his funding work on set theory, Cantor wrote:

> *A set is a gathering together into a whole of definite, distinct objects of our perception or of our thought—which are called elements of the set.* – George Cantor (1895)

We then define a set the following way:

Definition 1 (Set)**.** A collection of *distinct* objects, considered as an object in its own right. We define a set one of two way (always using braces):

- In extension by listing all the elements in the set: $\{1, 2, 4\}$
- In intension by specifying the rule that all elements follow: $\{n : n \in \mathbb{N} \land (n \mod 2 = 0)\}$

This definition alone gives some properties of sets. By being able to distinguish elements in the set from one another we assert that elements have an identity and we can derive equality from there. We can also give the empty set $\varnothing$ as the set with no elements. Since two sets are equals if and only if they have precisely the same elements, the empty set is unique. The member relation is noted $e \in \mathcal{S}$ to indicate that e is an element of $\mathcal{S}$. We note $\mathcal{S} \subset \mathcal{T} : (e \in \mathcal{S} \vDash e \in \mathcal{T}) \land \mathcal{S} \neq \mathcal{T}$, that a set $\mathcal{S}$ is a proper subset of a more general set $\mathcal{T}$.

We also define the union, intersection and difference as following:

- $\mathcal{S} \cup \mathcal{T} : \{e : e \in \mathcal{S} \lor e \in \mathcal{T}\}$

- $\mathcal{S} \cap \mathcal{T} : \{e : e \in \mathcal{S} \wedge e \in \mathcal{T}\}$
- $\mathcal{S} \setminus \mathcal{T} : \{e : e \in \mathcal{S} \wedge e \notin \mathcal{T}\}$

TODO: Axiom of fundation

1.1.1.3 Description Logics

On of the most standard and flexible way of representing knowledge for databases is by using ontologies. They are based mostly on the formalism of Description Logics (DL). It is common when using DLs to store statements into three boxes (Baader 2003):

- The TBox for terminology (statements about types)
- The RBox for rules (statements about properties) (Bürckert 1994)
- The ABox for assertions (statements about individual entities)

These are used mostly to separate knowledge about general facts (intentional knowledge) from specific knowledge of individual instances (extensional knowledge). The extra RBox is for "knowhow" or knowledge about entities behaviour. It restrict usages of roles (properties) in the ABox. The terminology is often hierarchically ordered using a subsumption relation noted $\prec$. If we represent classes or type as a set of individuals then this relation is akin to the subset relation of set theory.

TODO: More and some examples ?

1.1.1.4 Graphs

Next in line, we need to define a few notion of graph theory.
Definition 2 (Graph)**.** A graph is a mathematical structure $g = (V, \mathcal{E})$ consisting of vertices V (also called nodes) and edges $\mathcal{E}$ (arcs) that links two vertices together. Each edge is basically a pair of vertices ordered or not depending if the the graph is directed or not.

In the following, the signed symbols only applies to dirrected graphs.

We provide graphs with an adjacence function ϕ over any vertex $v \in V$ such that:

- $\phi(v) = \{e \in \mathcal{E} : v \in e\}$
- $\phi^+(v) = \{\langle v, v' \rangle \in \mathcal{E} : v' \in V\}$
- $\phi^-(v) = \{\langle v', v \rangle \in \mathcal{E} : v' \in V\}$

And an incidence function using the same name over any edges $e = \langle v, v' \rangle \in \mathcal{E}$ such that:

- $\phi(e) = \langle v, v' \rangle$
- $\phi^-(e) = v$

- $\phi^+(e) = v'$

Properties of graphs are better explained relative to their *transitive cover* χ^* of a graph $g = (V, \mathcal{E})$ defined as follow:

- $\chi(g) = (V, \mathcal{E}') : \mathcal{E}' = E \cup \{\langle v_1, v_3 \rangle : \{\langle v_1, v_2 \rangle, \langle v_2, v_3 \rangle\} \subset \mathcal{E}\}$
- $\chi^2 = \chi \circ \chi$
- $\chi^n = \bigcirc_{i=0}^{n} \chi$
- $\chi^* = \bigcirc^{\infty} \chi$

Now we present two additional notions on graphs.
Definition 3 (Path)**.** We say that vertices v_1 and v_2 are *connected* if it exists a path from one to the other. Said otherwise, there is a path from v_1 to v_2 if and only if $\langle v_1, v_2 \rangle \in \mathcal{E}_{\chi^*(g)}$.

Similarly we define *cycles* as the existence of a path from a given vertex to itself. Some graphs can be strictly acyclical, enforcing the absence of cycles.

TODO: Needs diagrams

1.1.1.5 Hypergraphs

A generalization of graphs are **hypergraphs** where the edges are allowed to connect more than two vertices. An hypergraph is n*-uniform* when the edges are restricted to connect only n vertices together. In that regard, classical graphs are 2-uniform hypergraphs.

TODO: Diagram of hypergraph

Another variant of graphs are graphs with the special case where $\mathcal{E} \subset V$. This means that edges are allowed to connect to other edges. These kind of graphs are a generalization of hypergraphs. Informations about these kind of structures for knowledge reare hard to come by and rely mostly on a form of "folk wisdom" where knowledge is rarely published and mostly transmitted orally during lessons. One of the closest information is this forum post (Kovitz 2018) that associated this type of graph to port graphs (Silberschatz 1981). Additional information was found in the form of a contribution of Vepstas (2008) on an encyclopedia article about hypergraphs. In that contribution, he says that a generalization of hypergraph allowing for edge-to-edge connections violates the axiom of fundation of ZFC by allowing edge-loops.

TODO: Diagram of edge-loops

This shows the limits of standard mathematics especially on the field of knowledge representation. Some strucutres needs higher dimensions than allowed by the one dimensional structure of ZFC and FOL.

However, it is important to not be mistaken: such non-standard set theories are more general than ZFC and therefore contains ZFC as a subset. All is a matter of restrictions.

1.1.1.6 Sheaf

In order to understand sheaves, we need to present a few auxiliary notions. Most of these definitions are adapted from (Vepštas 2008). The first of which is a seed.

Definition 4 (Seed)**.** A seed correspond to a vertex along with the set of adjacent edges. Formally we note a seed $(v, \phi_g(v))$ that means that a seed build from vertex v in the graph g contains a set of adjacent edges $\phi_g(v)$. We call the vertex v the *germ* of the seed.

Now we can build a kind of partial graphs from seeds called sections.

Definition 5 (Section)**.** A section is a set of seeds that have their common edges connected. This means that if two seeds have an edge in common connecting both germs, then the seeds are connected in the section and the edges are merged.

This tool was originally mostly meant for big data and categorization over large graphs. This is the reason for the next notion.

Definition 6 (Graph Quotient)**.** A quotient over a graph is the act of reducing a subgraph into a node while preserving the external connections. All internal structure becomes ignored and the subgraph now acts like a regular node.

Porting that notion to sections instead of graphs allows us to define stalks.

Definition 7 (Stalk)**.** Given a projection $p : V \to V'$ over the germs of a section s, the stalk above the vertex $v' \in V'$ is the quotient of all seeds that have their germ follow $p(v) = v'$.

Now we can add the final definition of sheaves.

Definition 8 (Seaf)**.** A seaf is a collection of sections, together with a projection function p and gluing axioms that the projection should respect depending on the application.

TODO: Diagrams and examples all the way.

1.1.2 Grammar and Parsing

Grammar is an old tool that used to be dedicated to liguists. With the funding works of Chomsky and his Contex-Free Grammars (CFG), these tools became available to mathematicians and shortly after to computer scientists.

A CFG is a formal grammar that aims to generate a formal language given a set of hierarchical rules. Each rule is given a symbol as a name. From any finite input of text in a given alphabet, the grammar should be able to determine if the input is part of the language it generates.

In computer science, popular meta-language called BNF was created shortly after Chomsky's work on CFG. The syntax is of the following form :

```
<rule> ::= <other_rule> | <terminal_symbol> | "literals"
```

A terminal symbol is a rule that does not depend of any other rule. It is possible to use recursion, meaning that a rule will use itself in its definition. This actually allows for infinite languages. Despite its expressive power, BNF is often used in one of its extended form.

In this context, we present a widely used form of BNF syntax that is meant to be human readable despite not being very formal. We add the repetition operators * and + that respectively repeat 0 and 1 times or more the preceeding expression. We also add the negation operator ~ that matches only if the following expression does not match. We also add parentesis for grouping expression and brackets to group literals.

A regular grammar is static, it is set once and for all and will always produce the same language. In order to be more flexible we need to talk about dynamic grammars and their associated tools.

One of the main tool for both static and dynamic grammar, is a parser. It is the program that will interprete the input into whatever usage it is meant for. Most of the time, a parser will transform the input into another similarily structured language. It can be a storage inside objects or memory, or compiled into another format, or even just for syntax coloration. Since a lot of usage requires the same kind of function, a new kind of tool emerged to make the creation of a parser simpler. We call those tools parser or compiler generators (Paulson 1982). They take a grammar description as input and gives the program of a parser of the generated language as an output.

For dynamic grammar, these tools can get more complicated. There are a few ways a grammar can become dynamic. The most straight-forward way to make a parser dynamic is to introduce code in the rule handling that will tweak variables affecting the parser itself (Souto *et al.* 1998). This allows for handling context in CFG without needing to rewrite the grammar.

Another kind of dynamic grammar are grammar that can modify themselves. In order to do this a grammar is valuated with reified objects representing parts of itself (Hutton and Meijer 1996). These parts can be modified dynamically by rules as the input gets parsed (Renggli *et al.* 2010; Alessandro and Piumarta 2007). This approach uses Parsing Expression Grammars (PEG)(Ford 2004) with Packrat parsing that Packrat parsing backtracks by ensuring that each production rule in the grammar is not tested more than once against each position in the input stream (Ford 2002). While PEG are easier to implement and more efficient in practice than their classical conterparts (Loff *et al.* 2018; Henglein and Rasmussen 2017), it offset the computation load in memory making it actually less efficient in general (Becket and Somogyi 2008).

Some tools actually just infer entire grammars from inputs and softwares (Höschele and Zeller 2017; Grünwald 1996). However, these kind of approaches require a lot of input data to perform well. They also simply provide the grammar after expensive computations.

1.1.3 Ontologies and their Languages

Most AI problems needs a way to represent data. The classical way to represent knowledge has been more and more specialized for each AI community. Each their Domain Specific Language (DSL) that neatly fit the specific use it is intended to do. There was a time when the branch of AI wanted to unify knowledge description under the banner of the "semantic web". Today this effort led to a way to standardize web service description but haven't reached the consensus the founders had hoped for. From numerous works, a repeated limitation of the "semantic web" seams to come from the languages used. In order to guarantee performance of generalist inference engines, these languages have been restricted so much that they became quite complicated to use and quickly cause huge amounts of recurrent data to be stored because of some forbidden representation that will push any generalist inference engine into undecidability.

TODO: OWL is more expressive ?!? (Van Harmelen *et al.* 2008, vol. 1 p850)

The most basic of these language is perhaps RDF Turtle (Beckett and Berners-Lee 2011). It is based on tripples with an XML syntax and has a graph as its knowledge structure (Klyne and Carroll 2004). A RDF graph is a set of RDF triples $\langle sub, pro, obj\rangle$ which fields are respectively called subject, property and object. It can also be seen as a partially labeled directed graph $(N, \mathcal{E})$ with N being the set of RDF nodes and $\mathcal{E}$ being the set edges. This graph also comes with an incomplete $label : (N \cup \mathcal{E}) \to URI$ relation. Nodes without an URI are called blank nodes. It is important that, while not named, blank nodes have a distinct internal identifier from one another that allows to differentiate them.

Inference in RDF has been shown to be trivially undecidable (Motik 2007). However, this language is probably the most permissive and therefore expressive language of the "semantic web". Even with this expressivity, several works still deems it as not expressive enough, mostly due to the lack of classical constructs like lists, parameters and quantifiers that don't fit the triple representation of RDF.

One of the ways which have been explored to overcome these limitations is by adding a 4^{th} field in RDF. This field is meant for context and anotations. This field is used for informations about any statement represented as a triple, such as access rights, belief and probabilities, or most of the time the source of the data (Tolksdorf *et al.* 2004). One of the other use of the fourth field of RDF is to reify statements (Hernández *et al.* 2015). Indeed by identifying each statement, it becomes possible to efficiently for statements about statements.

A completely different approach is done by Hart and Goertzel (2008) in his framework for Artificial General Intelligence (AGI) called OpenCog. The structure of the knowledge is based on a rhizome, collection of trees, linked to one another. This structure is called Atomspace. Each vertex in the tree is an atom, leaf-vertexes are nodes, the others are links. Atoms are immutable, indexed objects. They can be given values that can be dynamical and, since they are not part of the rhizome, are an order of magnitude faster to access. Atoms and values alike are typed.

The goal of such a structure is to be able to merge concepts from widely different domains of AI. The major drawback being that the whole system is very slow compared to pretty much any domain specific software.

1.2 WORLD

As we have seen, most existing knowledge description systems have a common drawback: they are static. This means that they are either too inefficient or too specific. To fix this issue, a new knowledge representation system must be presented. The goal is to make a minimal language class that can adapt to its use to become as specifc as needed. If they become specific they must start from a generic base. Since that base language must be able to evolve to fit the most cases possible, it must be neutral and simple.

To summarize, that language must maximize the following criteria:

1. **Neutral**: Must not assume preferences and be localization .
2. **Permissive**: Must allow as many data representation as possible.
3. **Minimalist**: Must have the minimum amount of base axioms and as little native notions as possible.
4. **Adaptive**: Must be able to react to user input and be as flexible as possible.

In order to respect these requirements, we developed a framework for knowledge description.

WORLD Offers Reflexive Languages Definition. It is our answer to these criteria. WORLD is inspired by RDF Turtle and Description Logic.

1.2.1 Knowledge Structure

WORLD extends the RDF graphs by adding another label to the edges of the graph to uniquely identify each statement. This basically turns the system into a quadruple storage even if this forth field is transparent to the user.

Axiom (Structure). A WORLD graph is a set of statements that transparently include their own identity. The closest representation of the underlying structure of WORLD is as follow :

$$W = (E, S) \,:\, S = \left\{ s = \langle e_{sub}, e_{pro}, e_{obj} \rangle \,:\, s \in D \vDash s \wedge D \right\}$$

with:

- $e_{sub}, e_{obj} \in E$ being entities representing the *subject* and *object* of the *statement* s,
- $e_{pro} \in P$ being the *property* of the statement s,
- $D \subset S$ is the *domain* of the *world* W,
- $S, P \subset E$ with S the set of statements and P the set of properties,

This means that W is a graph with the set of entities E as vertices and the set of statements S as edges. This model also suppose that every statement s must be true if they belong to the domain D. This graph is a directed 3-uniform H-hypergraph.

TODO Correct name + sheaf !

1.2.1.1 Consequences

TODO RDF graph as a projection

The base knowledge structure is more than simply convenience. The fact statements have their own identity changes the degrees of freedom of the representation. RDF has a way to represent reified statements that are basically blank nodes with properties that provide information about the subject, property and object of a designated statement. The problem is that, such statements are very differently represented and need 3 regular statements just to define. Using the forth field, it becomes possible to make statements about *any* statements. It becomes also possible to express modal logic about statements or to express, various traits like the probability or the access rights of a statements.

The knowledge structure holds several restrictions on the way to express knowledge. As a direct consequence, we can add several theorems to the logic system underlying WORLD. The axiom of Structure is the only axiom of the system.

Theorem 1 (Identity). *Any entity is uniquely distinct from any other entity.*

This theorem comes from the definition of mathematical sets and the extensionality axiom of the Z theory. Indeed it is stated that a set is a unordered collection of distinct objects. Distinction is possible if and only if intrinsic identity is assumed.

Theorem 2 (Consistency). *Any statement in a given domain is consistent with any other statements of this domain.*

Consistency comes from the need for a coherent knowledge database and is often a requirement of such constructs. This theorem also is a consequence of the axiom of Structure: $s \in D \vDash s \wedge D$.
Theorem 3 (Uniformity)**.** *Any object in WORLD is an entity. Any relations in WORLD are restricted to* E*.*

This also means that relations are closed under E.

1.2.1.2 Native properties

Theorem 1 leads to the need for two native properties in the system : *equality* and *name*.

The **equality relation** $=: E \to E$, behaves like the classical operator. Since the knowledge database will be accessed through a text interface, we also need to add an explicit way to identify entities. This identification is done through the **name relation** $\nu : E \to String$ that affects a string literal to some entities. This lead us to introduce literals into WORLD that are also entities that have a native value (more on literal **LATER**).

The axiom of Structure puts a type restriction on property. Since it compartments E using various named subsets, we must adequately introduce an explicit type system into WORLD. That type system requires a **type relation** $\tau : E \to T$. That relation is complete as all entities have a type. Theorem 3 causes the set of entities to be universal. Type theory, along with Description Logic (DL), introduce a **subsumption relation** $\prec: T \to T$ as a partial ordering relation to the types. In our case, the entity type is the greatest element of the lattice formed by the set of types with the subsumption relation $(T, \prec)$.

The uniformity of WORLD also allows some very interesting meta-constructs. That is why we also introduce a **meta relation** $\mu^{\pm} : E \rightleftharpoons^{+}_{-} D$. This allows to create domain from certain entities and to encapsulate domains into entities. μ^{-} is for reification and μ^{+} is for abstraction.

From theorem 2, it is supposed that only some statements have to be true. This along with the name property holds inside a construct we call a scope. This lead us to introduce the **scope operator** $\sigma : E \to \mathbb{P}(E)$ that affects to each entity to all entities representing its scope. Scopes are delimited by the meta relation as each abstraction level is separate from one another.

From the principle of adaptability and allowed under the meta relation, we can introduce a way for entities to be parametered and therefore having instances of their own. In order to do that, we introduce the **parameter relation** $\rho(e) : E \to E$. This relation

In figure 1.1, we present all the native relations along with their domains and most subsets of E.

FIXME

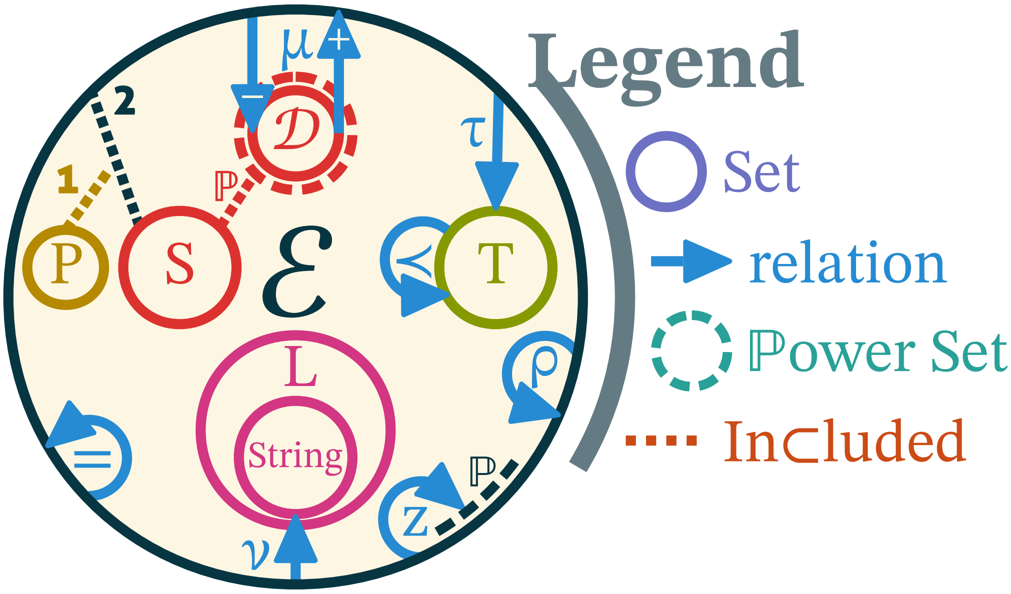

Figure 1.1: Venn diagram of subsets of E along with their relations. Dotted lines means that the sets are defined a subsets of the wider set.

1.2.2 Dynamic Grammar

Since we need to respect the requirements of the problem, the RDF syntax cannot be used to express the knowledge. Indeed, RDF states native properties as English nodes with a specific URI that isn't neutral. It also isn't minimalist since it uses an XML syntax that is so verbose it is not respected even in the documents that defines RDF because it is too confusing and complex. The XML syntax is also quite restrictive and cannot evolve dynamically to adapt to the usage.

So we need to define a new language that is minimalist and neutral. At the same time the language must be permissive and dynamical. These two goals are incompatible and will end up needing different solutions. So the solution to the problem is to actually define two languages that fit the criteria : one minimalist and one adaptive. The issue is that we need not make a user learn two languages and the second kind of language must be very specific and that violates the principle of neutrality we try to respect.

The only solution is to make a mechanism to adapt the language as it is used. We start off with a simple framework W_0 that uses a grammar g_0.

1.2.2.1 The grammar g_0

The description of g_0 is pretty straightforward: it mostly is just a triple representation separated by whitespaces. The goal is to add a minimal syntax consistent with the axiom of Structure. In listing 1.1, we give a simplified version of g_0. It is written in a pseudo-BNF fashion. It is extended with the classical repetition operators `*` and `+` along with the negation operator `~`. All tokens have names in uppercase. We also add the following rule modifiers:

- `<~name>` are ignored for the parsing. However, the tokens are consumed and therefore acts like separator for the other rules.
- `<?name>` are inferred rules and tokens. They play a key role for the process of derivation explained in section 1.2.2.2.

```
<~COMMENT: <INLINE: "//" (~["\n", "\r"])*>
|    <BLOCK: "/*" (~["*/"])*> > //Ignored
<~WHITE_SPACE: " "|"\t"|"\n"|"\r"|"\f">
<LITERAL: <INT> | <FLOAT> | <CHAR> | <STRING>> //Java definition
<ID: <TYPE: <UPPERCASE>(<LETTERS>|<DIGITS>)* >
|    <ENTITY:  <LOWERCASE>(<LETTERS>|<DIGITS>)*>
| <SYMBOL: (~[<LITERALS>, <LETTERS>, <DIGITS>])*>>

<world> ::= <first> <statement>* <EOF>
```

```
10 <first> ::= <subject> <?EQUAL> <?SOLVE> <?EOS>
11 <statement> ::= <subject> <property> <object> <EOS>
12 <subject> ::= <entity>
13 <property> ::= <ID> | <?meta_property>
14 <object> ::= <entity>
15 <entity> ::= <ID> | <LITERAL> | <?meta_entity>
```

Listing 1.1: Simplified pseudo-BNF description for basic WORLD.

In order to respect the principle of neutrality, the language must not suppose of any regional predisposition of the user. There are few exception for the sake of convenience and performance. The first exception is that the language is meant to be read from left to right and have an occidentally biased `subject verb object` triple description. Another exception is for literals that use the same grammar as in classical Java. This means that the decimal separator is the dot (.). This could be fixed in later version using dynamical definitions (see **LATER**).

Even if sticking to the ASCII subset of characters is a good idea for efficiency, WORLD can work with UTF-8 and exploits the Unicode Character Database (UCD) for its token definitions (Unicode Consortium 2018a). This means that WORLD comes keyword free and that the definition of each symbol is left to the user. Each notion and symbol is inferred (with the exception of the first statement, see **LATER**).

In g_0, the first two token definitions are ignored. This means that comments and white-spaces will act as separation and won't be interpreted. Comments are there only for convenience since they do not serve any real purpose in the language. It was arbitrarily decided to use Java style comments. White-spaces are defined against UCD's definition of the separator category `Z&` (see Unicode Consortium 2018b, chap. 4).

Line 4 uses the basic Java definition for literals. In order to keep the Independence from any natural language, boolean literals are not natively defined (since they are English words).

Another aspect of that language independence is found starting at line 5 where the definitions of <UPPERCASE>, <LOWERCASE>, <LETTERS> and <DIGITS> are defined from the UCD (respectively category `Lu`, `Ll`, `L&`, `Nd`). This means that any language's upper case can be used in that context. For performance and simplicity reasons we will only use ASCII in our examples and application.

The rule at line 10 is used for the definition of three tokens that are important for the rest of the input. <EQUAL> is the symbol for equality and <SOLVE> is the symbol for the solution quantifier (and also the language pendant of μ^-). The most useful token <EOS> is used as a statement delimiter. This rule also permits the inclusion of other files if a string literal is used as subject. The underlying logic of this first statement will be presented in **LATER**.

At line 11, we can see one of the most defining feature of W_0: statements. The input is nothing but a set of statements. Each components of the statements are entities. We defined two specific rules for the subject and object to allow for eventual run-time modifications. The property is more restricted in order to guarantee the non-ambiguity of the grammar.

1.2.2.2 Grammar derivation

The syntax we described is only valid for g_0. As long as the input is conforming to these rules, the framework keeps the W_0 behavior. In order to access more features, one need to break a rule. We add a second outcome to paring failure : **derivation**. There are several kind of possible violations that will interrupt the normal parsing of the input :

- Violations of the `<first>` statement rule : This will cause a fatal error.
- Violations of the `<statement>` rule : This will cause a derivation if an unexpected additional token is found instead of `<EOS>`. If not enough tokens are present, a fatal error is triggered.
- Violations of the secondary rules (`<subject>`, `<entity>`, ...) : This will give a fatal error except if there is also an excess of token in the current statement which will cause derivation to happen.

Derivation will cause the current input to be analyzed by a set of meta-rules. The main restriction of these rules is given in g_0: each statement must be expressible using a triple notation. This means that the goal of the meta-rules is to find an interpretation of the input that is consistent with this rule and to augment g_0 by adding an expression to any `<meta_*>` rules. If the input has less than 3 entity for a statement then the parsing fails. When there is extra input in a statement, there is a few ways the infringing input can be reduced back to a triple.

1.2.2.2.1 Containers The first meta-rule is to infer a container. A container is delimited by at least a left and right delimiter (they can be the same symbol) and an optional middle delimiter. We infer the delimiters using the algorithm 1.

The function sortedDelimiters at line 11 is used to generate every ordered possibility and sort them using a few criteria. The default order is possibilities grouped from left to right. All coupled delimiters that are mirror of each other following the UCD are preferred to other possibilities.

Checking the result of the choice is very important. At line 12 a function checks if the delimiters allows for triple reduction and enforce restrictions. For example, a property cannot be wrapped in a container. This is done in order to avoid a type mismatch later in the interpretation.

Once the inference is done, the resulting calls to inferDelimiter will add the rules listed in listing 1.2 to g_0. This function will create a `<container>` rule and add it to the definition of `<meta_entity>`. Then it will create a rule for the container named after the UCD name of the left delimiter (searching the

Algorithm 1 Container meta-rule

```
function container(Token current)
    lookahead(current, EOS)                                  ▷ Populate all tokens of the statement
    for all token in horizon do
        if token is a new symbol then delimiters.append(token)
    if length(delimiters) <2  then
        if coherentDelimiters(horizon, delimiters[0])  then
            inferMiddle(delimiters[0])                       ▷ New middle delimiter in existing container
            return Success
        return Failure
    while length(delimiters) > 0 do
        for all (left, middle, right) in sortedDelimiters(delimiters) do
            if coherentDelimiters(horizon, left, middle, right)  then
                inferDelimiter(left, right)
                inferMiddle(middle)                          ▷ Ignored if null
                delimiters.remove(left, middle, right)
                break
        if length(delimiters) stayed the same  then return Success
    return Success
```

`NamesList.txt` file for an entry starting with "left" and the rest of the name or defaulting to the first entry). Those rules are added as a conjunction list to the rule `<container>`. It is worthy to note that the call to inferMiddle will add rules to the token `<MIDDLE>` independently from any container and therefore, all containers share the same pool of middle delimiters.

```
1 <meta_entity> ::= <container>
2 <container> :: = <parentesis> | …
3 <parentesis> ::= "(" [<naked_entity>] (<?MIDDLE> <naked_entity>)* ")"
4 <naked_entity> ::= <statement> | <entity>
```

Listing 1.2: Rules added to the current grammar for handling new container for parenthesis

The rule at line 4 is added once and enables the use of meta-statements inside containers. This notion will be detailed in **LATER**.

1.2.2.2.2 Parameters

If the previous rule didn't fix the parsing of the statement, we continue with the following meta-rule. Parameters are containers that are used after an entity. Every container can be used as parameters. We detail the analysis in algorithm 2.

Algorithm 2 Parameter meta-rule

```
function parameter(Entity[] statement)
    reduced = statement
    while length(reduced) >3 do
        for i from 0 to length(reduced) - 1 do
            if name(reduced[i]) not null and
type(reduced[i+1]) = Container and
coherentParameters(reduced, i) then
                param = inferParameter(reduced[i], reduced[i+1])
                reduced.remove(reduced[i], reduced[i+1])
                reduced.insert(param, i)                ▷ Replace parameterized entity
                break
        if length(statement) stayed the same  then return Success
    return Failure
```

The goal is to match extra containers with the preceding named entity. The container is then combined with the preceding entity into a parameterized entity.

The call to inferParameter will add the rule in listing 1.3, replacing `<?container>` with the name of the container used (for example `<parenthesis>`).

```
<meta_entity> ::= <ID> <?container>
<meta_property> ::= <ID> <?container>
```

Listing 1.3: Rules added to the current grammar for handling parameters

1.2.2.2.3 Operators A shorthand for parameters is the operator notation. It allows to affect a single parameter to an entity without uing a container. It is most used for special entities like quantifiers or modificators. This is why once used the parent entity take a polymorphic type, meaning that type inference will not issue errors for any usage of them. Details of the way the operators are reduced is exposed in algorithm 3.

From the call of inferOperator, comes new rules explicated in listing 1.4. The call also add the operator entity to an inferred token `<OP>`.

```
<meta_entity> ::= <?OP> <ID>
```

Algorithm 3 Operator meta-rule

```
function operator(Entity[] statement)
    reduced = statement
    while length(reduced) >3 do
        for i from 0 to length(reduced) - 1 do
            if name(reduced[i]) not null and
name(reduced[i+1]) not null and
(name(reduced[i]) is a new symbol or
reduced[i] has been parameterized before) and
coherentOperator(reduced, i) then
                op = inferOperator(reduced[i], reduced[i+1])
                reduced.remove(reduced[i], reduced[i+1])
                reduced.insert(op, i)                    ▷ Replace parameterized entity
                break
        if length(statement) stayed the same then return Success
    return Failure
```

```
<meta_property> ::= <?OP> <ID>
```

Listing 1.4: Rules added to the current grammar for handling operators

If all meta-rules fail, then the parsing fails and returns an error to the user.

1.2.3 Contextual Interpretation

While parsing another important part of the processing is done after the success of a grammar rule. The grammar in WORLD is valuated, meaning that each rule has to return an entity. A set of functions are used to then populate the database with the right entities or retrieve an existing one that correspond to what is being parsed.

When parsing the rules `<entity>` and `<property>` will ask for the creation or retrieval of an entity. This mechanism will use the name of the entity and its type to retrieve an entity with the same name in a given scope.

1.2.3.1 Naming and Scope

When parsing an entity by name, the system will first request for an existing entity with the same name. If such an entity is retrieved it is returned instead of creating a new one. The validity of a name is limited by the notion of scope previously discussed.

Scopes are delimited by containers and statements. This local context is useful when wanting to restrict the scope of the declaration of an entity. The main goal of such restriction is to allow for a similar mechanism as the RDF namespaces. This makes the use of variable (RDF blank nodes) possible.

The scope of an entity have three special values :

- Variable: This scope restricts the scope of the entity to only the other entities in its scope.
- Local: This scope means that the parsing is still populating the scope of the entity. Its scope is limited to the currently parsing statement.
- Global: This scope means the name has no scope limitation.

The scope of an entity also contains all its parent entities, meaning all container or statement the entity is part of. This is used when chosing between the special values of the scope. The process is detailed in algorithm 4.

Algorithm 4 Determination of the scope of an entity

function scope(Entity e)
    Entity[] self = []
    if $\tau(e) = S$ **then**
        for all $i \in \phi(e)$ **do** self.append(self(i)) ▷ *Adding scopes nested in statement* e
    for all $i \in \mu^{-}(e)$ **do** self.append(self(i)) ▷ *Adding scopes nested in container* e
    if $\exists \rho(e)$ **then**
        Entity[] param = scope($\rho(e)$)
        for all $i \in$ param **do** param.remove(scope(i)) ▷ *Remove duplicate scopes from parameters*
        for all $i \in$ param **do** self.append(self(i)) ▷ *Adding scopes from paramters of* e
    $\sigma(e) \leftarrow$ self
    if GLOBAL $\notin \sigma(e)$ **then** $\sigma(e) \leftarrow \sigma(e) \cup$ {LOCAL}
return self
function self(Entity e)
    Entity[] self = []
    if LOCAL $\in \sigma(e)$ **then**
        for all $i \in \sigma(e)$ **do**
            if $\exists p \in E : \rho(p) = i$ **then** ▷ *e is already a parameter of another entity* p
                $\sigma(e) \leftarrow \sigma(e) \setminus$ {LOCAL}
                $\sigma(p) \leftarrow \sigma(p) \cup \sigma(e)$
                $\sigma(e) \leftarrow \sigma(e) \cup$ {VARIABLE, p}
        self.append(e)
        self.append($\sigma(e)$)
return self

The process happens for each entity created or requested by the parser. If a given entity is part of any other entity, the enclosing entity is added to its scope. When an entity is enclosed in any entity while already being a parameter of another entity, it becomes a variable.

1.2.3.2 Instanciation identification

When a parameterized entity is parsed, another process starts to identify if a compatible instance already exists. From theorem 1, it is impossible for two entity to share the same identifier. This makes mandatory to avoid creating an entity that is equal to an existing one. Given the order of which parsing is done, it is not always possible to determine the parameter of an entity before its creation. In that case a later examination will merge the new entity onto the older one and discard the new identifier.

1.2.4 Structure as a Definition

1.2.5 Extended Inference Mechanisms

1.3 Perpectives

1.3.1 Literal definition using Peano's axioms

1.3.2 Advanced Inference

2 General Planning Framework

2.1 Existing Languages and Frameworks

État de l'art

2.2 Taxonomy

2.2.1 Action type

(Définition)

2.2.2 Plan type

(Définition)

2.2.3 Problem type

(Définition)

2.3 Color

(Framework)

3 Online and Flexible Planning Algorithms

3.1 Existing Algorithms

État de l'art

3.2 Lollipop

3.2.1 Operator Graph

3.2.2 Negative Refinements

3.2.3 Usefullness Heuristic

3.2.4 Algorithm

3.2.5 Theoretical and Empirical Results

3.3 HEART

3.3.1 Domain Compilation

3.3.2 Abstraction in POP

3.3.3 Planning in cycle

3.3.4 Properties of Abstract Planning

3.3.5 Computational Profile

3.4 Planning Improvements

3.4.1 Heuristics using Semantics

3.4.2 Macro-Action learning

3.5 Recognition

3.5.1 Existing approcahes

3.5.2 Rico

3.5.2.1 Probabilities and approximations

We define a probability distribution over dated states of the world. That distribution is in part given and fixed and the rest needs computation. **TODO : that's super bad...**

Here is the list of given prior probabilities and asumptions :

- $P(O) = \prod_{o \in O} P(o)$
- $P(\mathcal{G}) = \sum_{G \in \mathcal{G}} P(G) = 1$ since we assume that the agent must be pursuing one of the goals.

- $P(G|\pi) = 1$ for a plan π appliable in I that achieves G.

From dirrect application of Bayes theorem and the previous assomptions, we have :

$$P(\pi|O) = \frac{P(O|\pi)P(\pi)}{P(O)} = \frac{P(O|\pi)P(\pi|G)P(G)}{P(O)} \tag{3.1}$$

$$P(G|O) = \frac{P(O|G)P(G)}{P(O)} \tag{3.2}$$

From the total probability formula :

$$P(O|G) = \sum_{\pi \in \Pi_G} P(O|\pi)P(\pi|G) \tag{3.3}$$

$$P(O|G) = \sum_{\pi \in \Pi_G} P(O|\pi)P(\pi|G) \tag{3.4}$$

4 Conclusion

Apendix

References

Alessandro, W., and I. Piumarta
OMeta: An object-oriented language for pattern matching, *Proceedings of the 2007 symposium on Dynamic languages*, 2007.

Baader, F.
The description logic handbook: Theory, implementation and applications, Cambridge university press, 2003.

Backus, J. W.
The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference, *Proceedings of the International Comference on Information Processing, 1959*, 1959.

Becket, R., and Z. Somogyi
DCGs+ memoing= packrat parsing but is it worth it?, *International Symposium on Practical Aspects of Declarative Languages*, Springer, 2008, 182–196.

Beckett, D., and T. Berners-Lee
Turtle - Terse RDF Triple Language, W3C Team Submission W3C, March 2011.

Bürckert, H.-J.
Terminologies and rules, *Workshop on Information Systems and Artificial Intelligence*, Springer, 1994, 44–63.

Cantor, G.
On a Property of the Class of all Real Algebraic Numbers., *Crelle's Journal for Mathematics*, 77, 258–262, 1874.

Cantor, G.
Beiträge zur Begründung der transfiniten Mengenlehre, *Mathematische Annalen*, 46 (4), 481–512, 1895.

Chomsky, N.
Three models for the description of language, *IRE Transactions on information theory*, 2 (3), 113–124, 1956.

Ciesielski, K.
Set Theory for the Working Mathematician, Cambridge University Press, August 1997.

Ford, B.
Packrat parsing:: Simple, powerful, lazy, linear time, functional pearl, *ACM SIGPLAN Notices*, ACM, 2002, 37,36–47.

Ford, B.
Parsing expression grammars: A recognition-based syntactic foundation, *ACM SIGPLAN Notices*, ACM, 2004, 39,111–122.

Fraenkel, A. A., Y. Bar-Hillel, and A. Levy
Foundations of set theory, vol. 67Elsevier, 1973.

Grünwald, P.
A minimum description length approach to grammar inference, in S. Wermter, E. Riloff, and G. Scheler (eds.), *Connectionist, Statistical and Symbolic Approaches to Learning for Natural Language Processing*, Lecture notes in computer science; Springer Berlin Heidelberg, 1996, 203–216.

Hart, D., and B. Goertzel
Opencog: A software framework for integrative artificial general intelligence, *AGI*, 2008, 468–472.

Henglein, F., and U. T. Rasmussen
PEG parsing in less space using progressive tabling and dynamic analysis, *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, ACM, 2017, 35–46.

Hernández, D., A. Hogan, and M. Krötzsch
Reifying RDF: What works well with wikidata?, *SSWS@ ISWC*, 1457, 32–47, 2015.

Horrocks, I., P. F. Patel-Schneider, and F. V. Harmelen
From SHIQ and RDF to OWL: The Making of a Web Ontology Language, *Journal of Web Semantics*, 1, 2003, 2003.

Höschele, M., and A. Zeller
Mining input grammars with AUTOGRAM, *Proceedings of the 39th International Conference on Software Engineering Companion*, IEEE Press, 2017, 31–34.

Hutton, G., and E. Meijer
Monadic parser combinators, 1996.

Klyne, G., and J. J. Carroll
Resource Description Framework (RDF): Concepts and Abstract Syntax, Language Specification. Ed. B. McBride W3C, 2004.

Kovitz, B.
Terminology - What do you call graphs that allow edges to edges?, *Mathematics Stack Exchange*, January 2018.

Loff, B., N. Moreira, and R. Reis
The Computational Power of Parsing Expression Grammars, *International Conference on Developments in Language Theory*, Springer, 2018, 491–502.

Motik, B.
On the properties of metamodeling in OWL, *Journal of Logic and Computation*, 17 (4), 617–637, 2007.

Paulson, L.
A semantics-directed compiler generator, *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '82*, Albuquerque, Mexico: ACM Press, 1982, 224–233. doi:10.1145/582153.582178.

Renggli, L., S. Ducasse, T. Gîrba, and O. Nierstrasz
Practical Dynamic Grammars for Dynamic Languages, *Workshop on Dynamic Languages and Applications*, Malaga, Spain, 2010, 4.

Silberschatz, A.
Port directed communication, *The Computer Journal*, 24 (1), 78–82, January 1981. doi:10.1093/comjnl/24.1.78.

Souto, D. C., M. V. Ferro, and M. A. Pardo
Dynamic Programming as Frame for Efficient Parsing, *Proceedings SCCC'98. 18th International Conference of the Chilean Society of Computer Science (Cat. No.98EX212)(SCCC)*, November 1998, 68. doi:10.1109/SCCC.1998.730784.

Tolksdorf, R., L. Nixon, F. Liebsch, Minh NguyenD., and Paslaru BontasE.
Semantic web spaces, 2004.

Unicode Consortium
Unicode Character Database, *About the Unicode Character Database*, https://www.unicode.org/ucd/#Latest, June 2018a.

Unicode Consortium
The Unicode Standard, Version 11.0, Core specification 11.0Mountain View, CA, June 2018b.

Van Harmelen, F., V. Lifschitz, and B. Porter
Handbook of knowledge representation, vol. 1Elsevier, 2008.

Vepstas, L.
Hypergraph edge-to-edge, *Wikipedia*, May 2008.

Vepštas, L.
Sheaves: A Topological Approach to Big Data, 2008.