1 Introduction

In antiquity, philosophy, mathematics and logic were considered as a single discipline. Since Aristotle we have realized that the world is not just black and white but full of nuances and colors. The inspiration for this thesis comes from one of the most influential philosophers and scientists of his time: Alfred Korzibsky. He founded a discipline he called *general semantics* to deal with problems of knowledge representation in humans. Korzibsky then found that complete knowledge of reality being inaccessible, we had to abstract. This abstraction is then only similar to reality in its structure. In these pioneering works, we find notions similar to that of modern descriptive languages.

It is from this inspiration that this document is built. We ~~then~~ start, off the beaten track and away from computer science by a brief excursion into the world of mathematical and logical formalism. This makes it possible to formalize a language that allows to describe itself partially by structure and that evolves with its use. The rest of the work illustrates the possible applications through specific fields such as automatic planning and intention recognition.

pas de virgule

1.1 Motivations

The social skills of modern robots are rather poor. Often, it is that lack that inhibits human-robot communication and cooperation. Humans being a social species, they require the use of implicit social cues in order to interact comfortably with an interlocutor.

In order to enhance assistance to dependent people, we need to account for any deficiency they might have. The main issue is that the patient is often unable or unwilling to express their needs. That is a problem even with human caregivers as the information about the patient's intents needs to be inferred from their past actions.

This aspect of social communications often eludes the understanding of Artificial Intelligence (AI) systems. This is the reason why intent recognition is such a complicated problem. The primary goal of this thesis is to address this issue and create the formal foundations of a system able to help dependent people.

1.2 Problem

First, *what exactly is intent recognition ?* The problem is simple to express: finding out what other agents want to do before they do. It is important to distinguish between several notions. *Plans* are the sequence of actions that the agent is doing to achieve

a *goal*. This goal is a concrete explanation of the wanted result. However, the *intent* is more of a set of abstract goals, some of which may be vague or impossible (e.g. drink something, survive forever, etc.).

to match

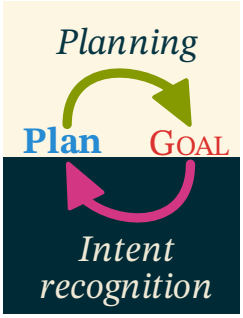
Some approaches use trivial machine learning methods, along with a hand-made plan library match observations to their most likely plan to use statistics. The issue with these common approaches is that they require an extensive amount of training data and need to be trained on each agent. This makes the practicality of such system quite limited. To address this issue, some works proposed hybrid approaches using logical constraints on probabilistic methods. These constraints are made to guide the resolution toward a more coherent solution. However, all probabilistic methods require an existing plan library that can be quite expensive to create. Also, plan libraries cannot take into account unforeseen or unlikely plans.

using

pas de virgule

A work from Ramırez and Geffner (2009), added an interesting method to solve this issue. Indeed, they noticed an interesting parallel between that problem with the field of automated planning. This analogy was made by using the Theory of mind (Baker *et al.* 2011), which states that any agent will infer the intent of other agents using a projection of their own expectations on the observed behaviors of the other agents.

and



This made the use of planning techniques possible to infer intents without the need for extensive and well-crafted plan libraries. Now only the domain of the possible actions, their effects and prerequisites are needed to infer the logical intent of an agent.

The main issue of planning for that particular use is computation time and search space size. This prevents most planners to make any decision before the intent is already realized and therefore being useless for assistance. This time constraint leads to the search of a real-time planner algorithm that is also expressive and flexible.

1.3 Contributions

In order to achieve such a planner, the first step was to formalize what is exactly needed to express a domain. Hierarchical and partially ordered plans gave the most expressivity and flexibility but at the cost of time and performance. This is why, a new formalism of knowledge representation was needed in order to increase the speed of the search space exploration while restricting it using semantic inference rules.

While searching for a knowledge representation model, we developed some prototypes using standard ontology tools but all proved to be too slow and inexpressive for that application. This made the design of a lighter but more flexible knowledge representation model, a requirement for planning domain representation.

Then the planning formalism has to be encoded using our general knowledge representation tool. Since automated planning has a very diverse ecosystem of approaches and paradigms, its standard, the Planning Domain Description Language (PDDL) needs use of various extensions. However, no general formalism has been given for PDDL

and some approaches often lack proper extensions (hierarchical planning, plan representation, etc.). This is why a new formalism is proposed and compared to the one used as standard of the planning community. as THE standard of

Then finally, a couple of planners were designed to attempt answering the speed and flexibility requirements of human intent recognition. The first one is a prototype that aims to evaluate the advantages of repairing plans to use several heuristics. The second is a more complete prototype derived from the first (without plan repairs), which also implements a Breadth-First Search (BFS) approach to hierarchical decomposition of composite actions. This allows the planner to provide intermediary plans that, while incomplete, are an abstraction of the result plans. This allows for anytime intent recognition probability computation using existing techniques of inverted planning.

1.4 Plan

In this document we will describe a few contributions from the new Structurally Expressive Language Framework for intent recognition. Each chapter builds on the previous one.

First we will present a new mathematical model that suits our needs. This axiomatic theory is used to create a model capable of describing all the mathematical notions required for our work. Chapter 2

In the third chapter, a new knowledge description system is presented as well as the associated grammar and inference framework. This system is based on triple representation to allow for structurally defined semantic. Chapter 3

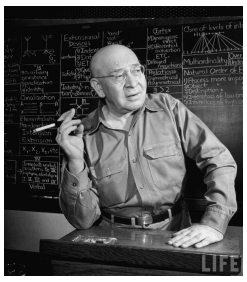
The chapter 4 is an introduction to automated planning along with a formal description of a general planner using appropriate search spaces and solution constraints. Chapter 4

The fifth chapter is an application of knowledge description to automated planning. This allows us to design a general planning framework that can express any existing type of domain. Existing languages are compared to our proposed approach. Chapter 5

Using this framework, two online planning algorithms are presented in chapter six: one that uses repairs on existing plans and one that uses hierarchical domains to create intermediary abstract plans. Chapter 6

The final chapter is about intent recognition and its link to planning. Existing works are presented as well as a technique called *inverted planning*. Chapter 7

2 Foundation and Tools



Alfred Korzybski (1933, ch. 4 pp. 58)

> *"A map* is not *the territory it represents, but, if correct, it has a* similar structure *to the territory, which accounts for its usefulness."*

Mathematics and logic are at the heart of all formal sciences, including computer science. The boundary between mathematics and computer science is quite blurry. Indeed, computer science is applied mathematics and mathematics are abstract computer science. Both cannot be separated when needing a formal description of a new model.

In mathematics, a *foundation* is an *axiomatic theory* that is consistent and well-defined. It can also be called a **model**. For a foundation to be generative of a subset of mathematics, it must define all the supported notions.

In this chapter, we define a new formalism as well as a proposed foundation that lies on the bases of the type theory and lambda calculus. With this formalism, we define the classical set theory (which is the foundation for classical mathematics). The contribution is mainly in the axiomatic system, and functional algebra. The rest is simply an explanation, using our formalism, of existing mathematical notions and structures commonly used in computer science. **This formalism is used for all the formulas later on this document.**

2.1 Existing Model Properties

Any knowledge must be expressed using an encoding support (medium) like a language. Natural languages are quite expressive and allow for complex abstract ideas to be communicated between individuals. However, in science we encounter the first issues with such a language. It is culturally biased and improperly convey formal notions and proof constructs. Indeed, natural languages are not meant to be used for rigorous mathematical proofs. This is one of the main conclusions of the works of Korzybski (1958) on "general semantics". The original goal of Korzybski was to pinpoint the errors that led humans to fight each other in World War I. He affirmed that the language is unadapted to convey information reliably about objective facts or scientific notions. There is a discrepancy between the natural language and the underlying structure of the reality.

conveyS

In the following sections we describe a few inherent properties of formalism and mathematical reasoning that are useful to consider when defining a theory.

2.1.1 Abstraction

> **abstraction** (n.d.): *The process of formulating generalized ideas or concepts by extracting common qualities from specific examples*

Collins English Dictionary (2014)

The idea behind abstraction is to simplify the representation of complex instances. This mechanism is at the base of any knowledge representation system. Indeed, it is unnecessarily expensive to try to represent all properties of an object. An efficient way to reduce that knowledge representation is to prune away all irrelevant properties while also only keeping the ones that will be used in the context. *This means that abstraction is a losy process* since information is lost when abstracting from an object.

Since this is done using a language as a medium, this language is a *host language*. Abstraction will refer to an instance using a *term* (also called symbol) of the host language. If the host language is expressive enough, it is possible to do abstraction on an object that is already abstract. The number of layers abstraction needed for a term is called its *abstraction level*. Very general notions have a higher abstraction level and we represent reality using the null abstraction level. In practice abstraction uses terms of the host language to bind to a referenced instance in a lower abstraction level. This forms a structure that is strongly hierarchical with higher abstraction level terms on top.

Example 1. We can describe an individual organism with a name that is associated to this specific individual. If we name a dog "Rex" we abstract a lot of information about a complex, dynamic living being. We can also abstract from a set of qualities of the specimen to build higher abstraction. For example, its species would be *Canis lupus familiaris* from the *Canidae* family. Sometimes several terms can be used at the same abstraction level like the commonly used denomination "dog" in this case.

Terms are only a part of that structure. It is possible to combine several terms into a *formula* (also called proposition, expression or statements).

2.1.2 Formalization

> **formal** (adj.): *Relating to or involving outward form or structure, often in contrast to content or meaning.*

American Heritage Dictionary (2011a)

A formalization is the act to make formal. The word "formal" comes from Latin *fōrmālis*, from *fōrma*, meaning form, shape or structure. This is the same base as for the word "formula". In mathematics and *formal sciences* the act of formalization is to reduce knowledge down to formulas. Like stated previously, a formula combines several terms. But a formula must follow rules at different levels:

- *Lexical* by using terms belonging in the host language.
- *Syntactic* as it must follow the grammar of the host language.
- *Semantic* as it must be internally consistent and meaningful.

The information conveyed from a formula can be reduced to one element: its semantic structure. Like its etymology suggests, a formula is simply a structured statement about terms. This structure holds its meaning. Along with using abstraction, it becomes possible to abstract a formula and therefore, to make a formula about other formulas should the host language allowing it.

Example 2. The formula using English "dog is man's best friend" combines terms to hold a structure between words. It is lexically correct since it uses English words and grammatically correct since it can be grammatically decomposed as (n. v. n. p. adj. n.). In that the (n.) stands for nouns (v.) for verbs (adj.) for adjectives and (p.) for possessives. Since the verb "is" is the third person singular present indicative of "be" and the adjective is the superlative of "good", this form is correct in the English language. From there the semantic aspect is correct too but that is too subjective and extensive to formalize here. We can also build a formula about a formula like "this is a common phrase" using the referential pronoun "this" to refer to the previous formula.

Any language is comprised of formulas. Each formula holds knowledge about their subject and state facts or belief. A formula can describe other formulas and even *define* them. However, there is a strong limitation of a formalization. Indeed, a complete formalization cannot occur about the host language. It is possible to express formulas about the host language but *it is impossible to completely describe the host language using itself* (Klein 1975). This comes from two principal reasons. As abstraction is a loose process one cannot completely describe a language while abstracting its definition. If the language is complex enough, its description requires an even more complex *metalanguage* to describe it. And even for simpler language, the issue stands still while making it harder to express knowledge about the language itself. For this we need knowledge of the language *a priori* and this is contradictory for a definition and therefore impossible to achieve.

When abstracting a term, it may be useful to add information about the term to define it properly. That is why most formal system requires a *definition* of each term using a formula. This definition is the main piece of semantic information on a term and is used when needing to evaluate a term in a different abstraction level. However, this is causing yet another problem.

2.1.3 Circularity

> **circularity** (n.d.): *Defining one word in terms of another that is itself defined in terms of the first word.*
>
> American Heritage Dictionary (2011b)

Circularity is one of the issues we explore in this section about the limits of formalization languages. Indeed, defining a term requires using a formula in the host language to express the abstracted properties of the generalization (Korzybski 1933). The problem is that most terms will have *circular* definitions.

Example 3. Using definitions from the American Heritage Dictionary (2011b), we can find that the term "word" is defined using the word "meaning" that is in turn defined using the term "word". Such circularity can happen to use an arbitrarily long chain of definition that will form a cycle in the dependencies.

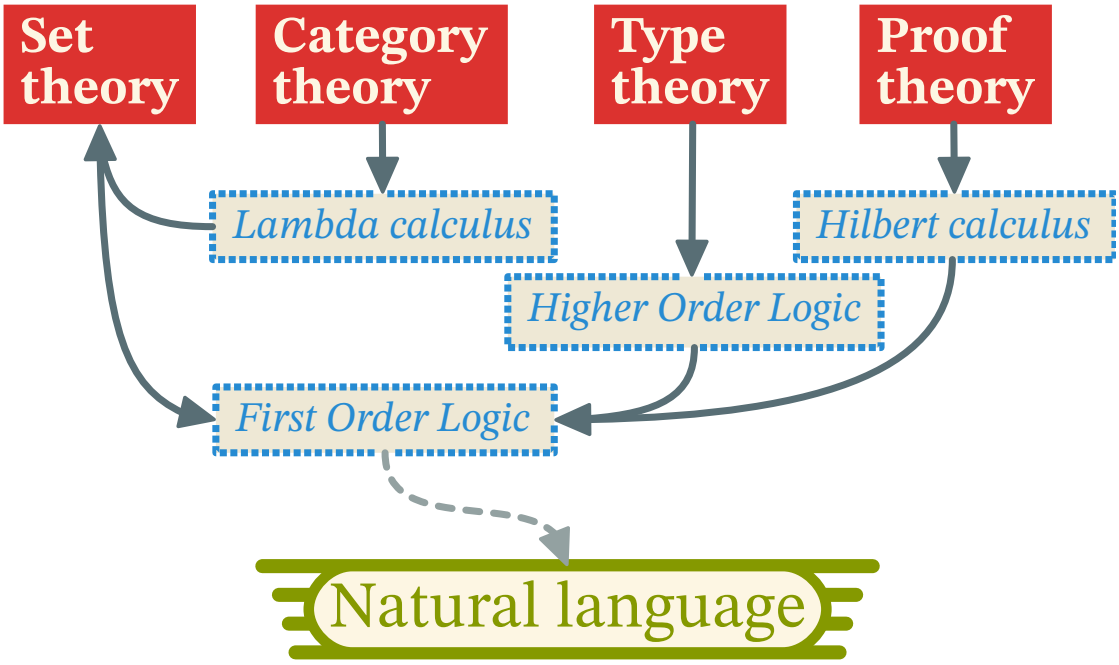


Figure 2.1: Dependency graph of the most common foundational mathematical theories and their underlying implicit formalism.

For example, we illustrate dependencies between some existing theories and their formalism in figure 2.1. Since a formalization cannot fully be self defined, another host language is generally used, sometimes without being acknowledged.

The only practical way to make some of this circularity disappear is to base on a natural language as host language for defining the most basic terms. This allows to acknowledge the problem in an instinctive way while being aware of it while building the theory.

2.2 Functional theory

We aim to reduce the set of axioms allowing to describe a model. The following theory is a proposition for a possible model that takes into account the previously described constraints. It is inspired by category theory (Awodey 2010), and typed lambda calculus (Barendregt 1984).

2.2.1 Category theory

This theory is based, as its name implies, on *categories*. A category is a mathematical structure that consists of two components:

- A set of **objects** that can be any arbitrary mathematical entities.
- A set of **morphisms** that are functional monomes. They are often represented as *arrows*.

Many definitions of categories exist (Barr and Wells 1990, vol. 49) but they are all in essence similar to this explanation. The best way to see the category theory is as a general theory of functions. Even if we can use any mathematical entity for the types of the components, the structure heavily implies a functional connotation.

2.2.2 Axioms

In this part, we propose a model based on functions. The unique advantage of it lays in its explicit structure that allows it to be fully defined. It also holds a coherent algebra that is well suited for our usage. This approach can be described as a special case of category theory. However, it differs in its priorities and formulation. For example, since our goal is to build a standalone foundation, it is impossible to fully specify the domain or co-domain of the functions and they are therefore weakly specified (Godel and Brown 1940).

Our theory is axiomatic, meaning it is based on fundamental logical proposition called axioms. Those form the base of the logical system and therefore are accepted without any need for proof. In a nutshell, axioms are true prior hypotheses.

The following axioms are the explicit base of the formalism. It is mandatory to properly state those axioms as all the theory is built on top of it.

$(=) = x \to x$

Axiom (Identity)**.** Let's the identity function be $=$ that associates every function to itself. This function is therefore transparent as by definition $=(x)$ is the same as x. It can be described by using it as **affectation** or aliases to make some expressions shorter or to define any function.

In the rest of the document, classical equality and the identity function will refer to the same notion.

That axiom implies that the formalism is based on *functions*. Any function can be used in any way as long as it has a single argument and returns only one value at a time (named image, which is also a function).

It is important to know that **everything** in our formalism *is* a function. Even notions such as literals, variables or set from classical mathematics are functions. This property is inspired by lambda calculus and some derived functional programming languages.

Using definition 1 and definition 2, we can note $(\to) = x \to (f(x) \to f)$

Axiom (Association)**.** Let's the term $\to$ be the function that associates two expressions to another function that associates those expressions. This special function is derived from the notation of *morphisms* of the category theory.

The formal definition uses currying to decompose the function into two. It associates a parameter to a function that takes an expression and returns a function.

Next we need to lay off the base definitions of the formalism.

2.2.3 Formalism definition

This functional algebra at the base of our foundation is inspired by *operator algebra* (Takesaki 2013, vol. 125) and *relational algebra* (Jónsson 1984). The problem with the operator algebra is that it supposes vectors and real numbers to work properly. Also, relational algebra, like category theory, presupposes set theory.

Here we define the basic notions of our functional algebra that dictates the rules of the formalism we are defining.

Definition 1 (Currying)**.** Currying is the operation named after the mathematician Haskell Brooks Curry (1958) that allows multiple argument functions in a simpler monoidal formalism. A monome is a function that has only one argument and has only one value, as the axiom of Association.

$(\!|f|\!) = (x \to (\!|f(x)|\!))$

The operation of *currying* is a function $(\!|\,|\!)$ that associates to each function f another function that recursively partially applies f with one argument at a time.

to x H gives

If we take a function h such that when applied to x gives the function g that takes an argument y, *unCurrying* is the function $\{\!|\,|\!\}$ so that $f(x,y)$ behaves the same way as $h(x)(y)$. We note $h = \{\!|f|\!\}$.

$\{\!|f|\!\} = \{\!|x, y \to f(x)(y)|\!\}^+$

Definition 2 (Application)**.** We note the application of f with an argument x as $f(x)$. The application allows to recover the image y of x which is the value that f associates with x.

$y = f(x)$

Along with Currying, function application can be used *partially* to make constant some arguments.

Definition 3 (Partial Application)**.** We call *partial application* the application using an insufficient number of arguments to any function f. This results in a function that has fewer arguments with the first ones being locked by the partial application. It is interesting to note that currying is simply a recursion of partial applications.

From now on we will note $f(x, y, z, ...)$ any function that has multiple arguments but will suppose that they are implicitly Curryied. If a function only takes two arguments, we can also use the infix notation e.g. $x\,f\,y$ for its application.

Example 4. Applying this to basic arithmetic for illustration, it is possible to create a function that will triple its argument by making a partial application of the multiplication function $\times(3)$ so we can write the operation to triple the number 2 as $\times(3)(2)$ or $\times(2,3)$ or with the infix notation 2×3.

$\times(3) = x \to 3 \times x$

Definition 4 (Null)**.** The *null function* is the function between nothing and nothing. We note it $\gg$.

$\gg = \gg \to \gg$

The notation $\gg$ was chosen to represent the association arrow ($\to$) but with a dot instead of the tail of the arrow. This is meant to represent the fact that it inhibits associations.

2.2.4 Literal and Variables

As everything is a function in our formalism, we use the null function to define notions of variables and literals.

$l =\gg\rightarrow l$

Definition 5 (Literal)**.** A literal is a function that associates null to its value. This consists of any function l written as $\gg\rightarrow l$. This means that the function has only itself as an immutable value. We call *constants* functions that have no arguments and have as value either another constant or a literal.

Example 5. A good example of that would be the yet to be defined natural numbers. We can define the literal 3 as $3 =\gg\rightarrow 3$. This is a function that has no argument and is always valued to 3.

$x = x \rightarrow\gg$

Definition 6 (Variable)**.** A variable is a function that associates itself to null. This consists of any function x written as $x \rightarrow\gg$. This means that the function requires an argument and has undefined value. Variables can be seen as a demand of value or expression and mean nothing without being defined properly.

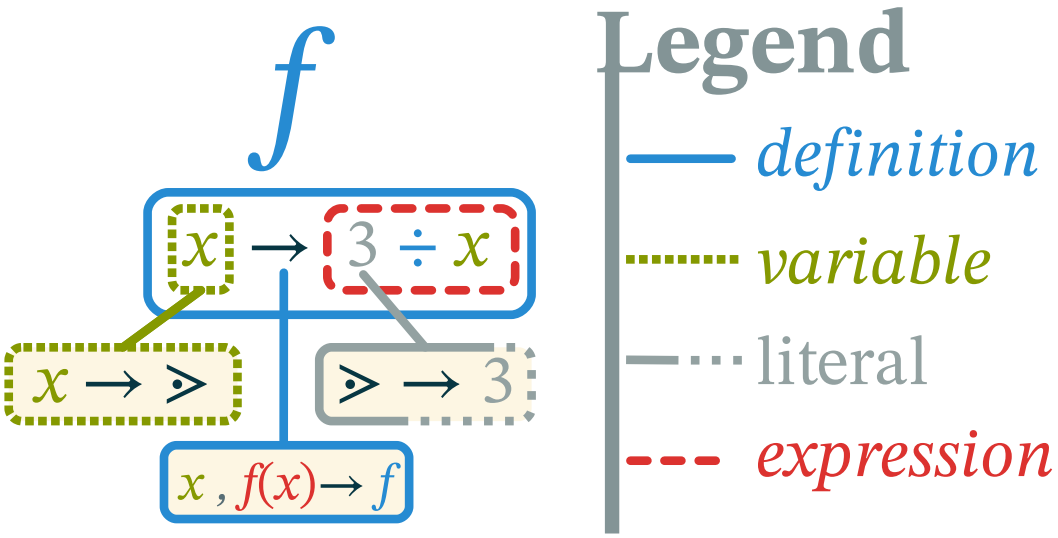


Figure 2.2: Illustration of basic functional operators and their properties.

Example 6. The function f defined in figure 2.2 associates its argument to an expression. Since the argument x is also a variable, the value is therefore dependent on the value required by x. In that example, the number 3 is a literal and $3 \div x$ is therefore an expression using the function $\div$.

An interesting property of this notation is that $\gg$ is both a variable and a constant. Indeed, by definition, $\gg$ is the function that associates $\gg\rightarrow\gg$ and fulfills both definitions.

When defining currying, we annotated it using the notation $(\!|f|\!) = f \rightarrow (x \rightarrow (\!|f(x)|\!))$. The obvious issue is the absence of stopping condition in that recursive expression.

While the end of the recursion doesn't technically happen, in practice from the way variables and literals are defined, the recursion chain either ends up becoming a variable or a constant because it is undefined when currying a nullary function.

2.2.5 Functional algebra

Inspired by relational algebra and by category theory, we present a functional algebra that fits our needs. The first operator of this algebra allows to combine several functions into one. This is very useful to merge the definition of two functions in order to specify more complex functions.

Definition 7 (Combination)**.** The *combination function* $\bowtie$ associates any two functions to a new function that is the union of the definition of **either** functions. If both functions are defined for any given argument, then the combination is undefined ($\gg$)

$\bowtie = (f, \gg \to f) \bowtie (\gg, f \to f)$

It is interesting to note that the formal definition of the combination is recursive. This means that it will be evaluated if any of the expression matches, decomposing the functions until one of them isn't defined or until nothing matches and therefore the result is $\gg$.

Combination is useful to define boolean constantly in first order logic (section 2.3.1).

Example 7. For two functions f_1 and f_2 that are defined respectively by:

- $f_1 = (1 \to 2) \bowtie (3 \to 4)$
- $f_2 = (2 \to 3) \bowtie (3 \to 5)$

the combination $f_3 = f_1 \bowtie f_2$ will behave as follows:

- $f_3(1) = 2$
- $f_3(2) = 3$
- $f_3(3) = \gg$

Definition 8 (Superposition)**.** The *superposition function* $\triangle$ associates any two functions to a new function. This function is what the definition of the two functions taken as arguments have in common. The resulting function associates $x \to y$ when **both** functions are superposing. We can say that the superposition is akin to a joint where the resulting function is defined when both functions have the same behavior.

$\triangle = (\bowtie) \bowtie (f, f \to f)$

Example 8. Reusing the functions of the previous example, we can note that $f_3 \triangle f_1 = 1 \to 2$ because it is the only association that both functions have in common. We can also say that $f_1 \triangle f_2 = \gg$ because these functions do not share any associations.

Superposition has a "negative" counterpart called a *subposition*. It allows to do the inverse operation of the super position. More intuitively, if the superposition is akin to the set "intersection", the subposition is the "difference" counterpart.

Definition 9 (Subposition)**.** The *subposition* is the function $\triangledown$ that associates any two functions f_1 and f_2 to a new function $f_1 \triangledown f_2$. The subposition will allow to "substract" associations from existing functions. The result removes the superposition from the first function definition.

$\triangledown = f_1, f_2 \to f_1 \bowtie (f_1 \triangle f_2)$

The subposition is akin to a subtraction of function where we remove everything defined by the second function to the definition of the first.

Example 9. From the previous example we can write $f_3 \triangledown f_2 = (1 \to 2)$. Since f_2 has the $2 \to 3$ associations in common with f_3 it is removed from the result.

This operation is more useful with the superposition as it can behave like so: $(f_1 \vartriangle f_2) \triangledown f_2 = f_1$. This allows to undo the superposition.

We can also note a few properties of these functions in table 2.1.

Table 2.1: Example properties of superposition and subposition

Formula	Description
$f \vartriangle f = f$	A function superposed to itself is the same.
$f \vartriangle \gg = \gg$	Any function superposed by null is null.
$f_1 \vartriangle f_2 = f_2 \vartriangle f_1$	Superposition order doesn't affect the result.
$f \triangledown f = \gg$	A function subposed by itself is always null.
$f \triangledown \gg = f$	Subposing null to any function doesn't change it.

These functions are intuitively the functional equivalent of the union, intersection and difference from set theory. In our formalism we will define the set operations from these.

The following operators are the classical operations on functions.

Definition 10 (Composition)**.** The *composition function* is the function that associates any two functions f_1 and f_2 to a new function such that: $f_1 \circ f_2 = x \to f_1(f_2(x))$.

Definition 11 (Inverse)**.** The *inverse function* is the function that associates any function to its inverse such that if $y = f(x)$ then $x = \bullet(f)(y)$.

We can also use an infix version of it with the composition of functions: $f_1 \bullet f_2 = f_1 \circ \bullet(f_2)$.

These properties are akin to multiplication and division in arithmetic.

Table 2.2: Example of function composition and inverse with their properties.

Formula	Description
$f \circ \gg = \gg$	This means that $\gg$ is the absorbing element of the composition.
$f \circ (=) = f$	Also, $=$ is the neutral element of the composition.
$\bullet(\gg) = \gg \wedge \bullet(=) = (=)$	This means that $\gg$ and $=$ are commutative functions.
$f_1 \circ f_2 \neq f_2 \circ f_1$	However, $\circ$ is not commutative.

From now on, we will use numbers and classical arithmetic as we had defined them. However, we consider defining them from a foundation point of view, later using set theory and Peano's axioms.

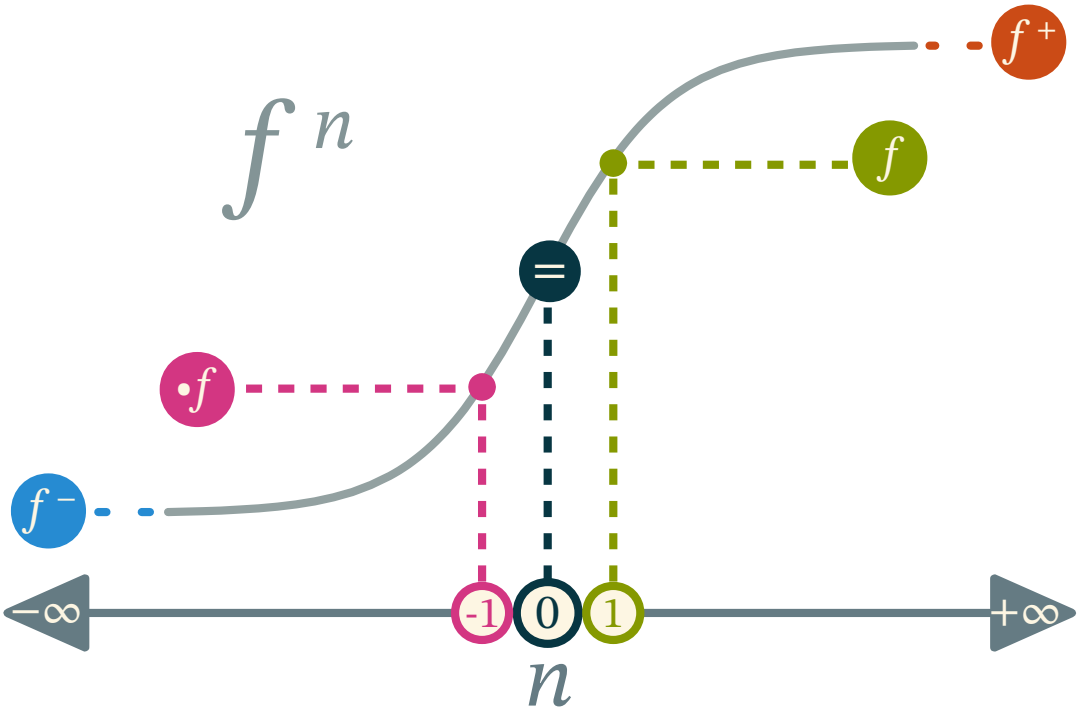


Figure 2.3: Illustration of how the functional equivalent of the power function is behaving with notable values (in filled circles)

In classical mathematics, the inverse of a function f is often written as f^{-1}. Therefore we can define the transitivity of the n^{th} degree as the power of a function such that $f^n = f^{n-1} \circ f$. Figure 2.3 shows how the power of a function is behaving at key values.

By generalizing the formula, we can define the *transitive cover* of a function f and its inverse respectively as $f^+ = f^{+\infty}$ and $f^- = f^{-\infty}$. This cover is the application of the function to its result infinitely. This is useful especially for graphs as the transitive cover of the adjacency function of a graph gives the connectivity function (see section 2.5).

We also call *arity* the number of arguments (or the currying order) of a function noted $|f|$.

2.2.6 Properties

A modern approach of mathematics is called *reverse mathematics* as instead of building theorems from axioms, we search the minimal set of axioms required by a theorem. Inspired by this, we aim to minimize the formal basis of our system as well as identifying the circularity issues, we provide a dependency graph in figure 2.4. We start with the axiom of Association at the bottom and the axiom of Identity at the top. Everything depends on those two axioms but drawing all the arrows makes the figure way less legible.

Then we define the basic application function () that has as complement the Currying (⫴) and unCurrying (❪❫) functions. Similarly, the combination (⋈) has the superposition (△) and the subpostion (▽) functions as complements. The bottom bound of the algebra is the null function (⩾) and the top is the identity function (=). Composition (∘) is the main operator of the algebra and allows it to have an inverse element as the inverse function (•). The composition function needs the application function in order to be constructed.

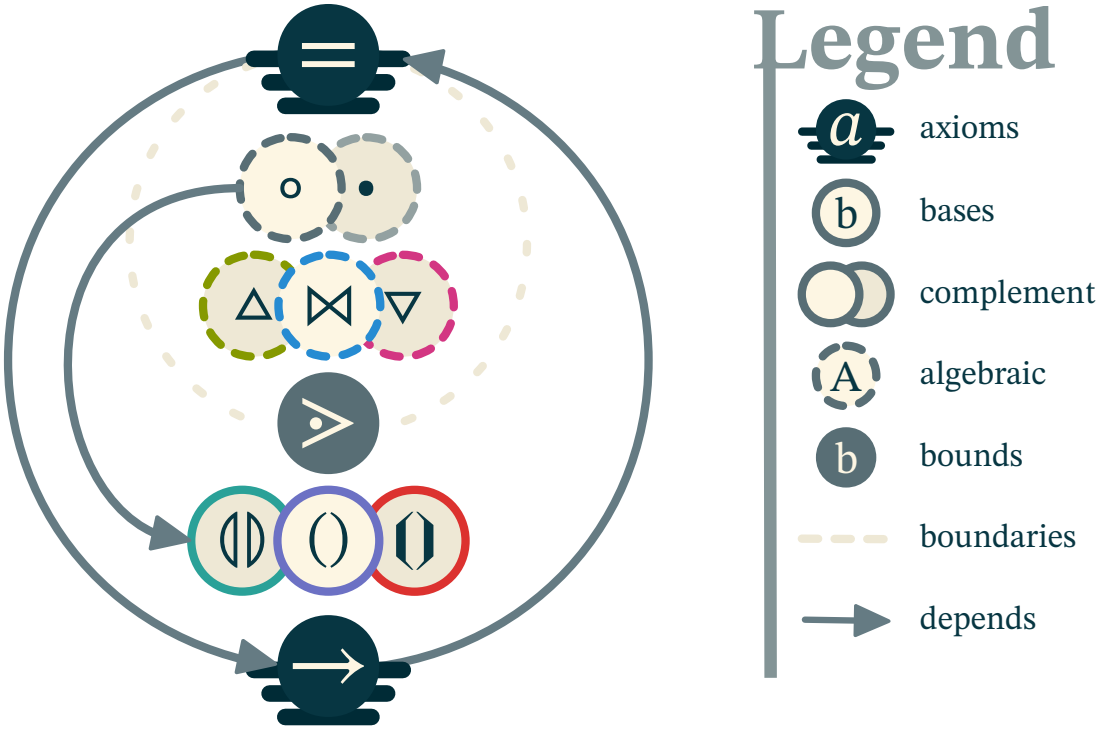


Figure 2.4: Dependency graph of notions in the functional theory

The algebra formed by the previously defined operations on functions is a semiring $(\mathbb{F}, \bowtie, \circ)$ with $\mathbb{F}$ being the set of all functions.

Indeed, $\bowtie$ is a commutative monoid having $\geqslant$ as its identity element and $\circ$ is a monoid with $(=)$ as its identity element.

Also the composition of the combination is the same as the combination of the composition. Therefore, $\circ$ (composition) distributes over $\bowtie$ (combination).

At last, using partial application composing with null gives null: $\circ(\geqslant) = ((\geqslant) \to \geqslant) = \geqslant$.

This foundation is now ready to define other fields of mathematics. We start with logic as it is a very basic formalism in mathematics.

2.3 Logic and reasoning

2.3.1 First Order Logic

In this section, we present First Order Logic (FOL). FOL is based on boolean logic with the two literals *true* (noted $\top$) and *false* (noted $\bot$).

A function noted $(?)$ that has as only values either $\top$ or $\bot$ is called a **predicate**. $\mathcal{D}(\bullet?) = \{\bot, \top\}$

We define the classical logic *entailment*, the predicate that holds true when a predicate (the conclusion) is true if and only if the first predicate (the premise) is true.

$$\vdash = (\bot, x \to \top) \bowtie (\top, x \to x)$$

Then we define the classical boolean operators $\neg$ *not*, $\wedge$ *and* and $\vee$ *or* as:

- $\neg = (\bot \to \top) \bowtie (\top \to \bot)$, the negation associates true to false and false to true.
- $\wedge = x \to ((\top \to x) \bowtie (\bot \to \bot))$, the conjunction is true when all its arguments are simultaneously true.
- $\vee = x \to ((\top \to \top) \bowtie (\bot \to x))$, the disjunction is true if all its arguments are not false.

The last two operators are curried function and can take any number of arguments as necessary and recursively apply their definition.

Another basic predicate is the **equation**. It is the identity function $=$ but as a binary predicate that is true whenever the two arguments are the same.

Functions that take an expression as parameters are called *modifiers*. FOL introduces a useful kind of modifier used to moralize expressions: *quantifiers*. The quantifiers take an expression and a variable as arguments. Classical quantifiers are also predicates: they restrict the values that the variable can take.

In the realm of FOL, quantifiers are restricted to individual variable (booleans) as follows:

- The *universal quantifier* $\forall$ meaning *"for all"*. $\forall = \S(\wedge)$
- The *existential quantifier* $\exists$ meaning *"it exists"*. $\exists = \S(\vee)$

2.3.2 Higher Order Logic

Higher Order Logic (HOL) is a class of logic formalism that supersedes FOL. It is, however, less well-behaved than FOL and is not as popular as a consequence. Indeed, HOL allows quantifiers to be applied to sets and even set of set (see section 2.4). This makes the expressivity of this kind of logic higher but also makes it harder to use and compute. setS

2.3.3 Modal logic

Section 3.6 will illustrate on an example.

Even bigger than HOL is modal logic. In that logic, quantifiers can be applied to *anything*. The most interesting feature of modal logic is quantifying expressions themselves. This allows for *modality* of statements such as their likelihood, context or to even ask for information.

Using that kind of logic, we can also add some less used quantifiers such as:

$\exists! = \S(= (1) \circ +)$
$\nexists = \S(\neg \circ \wedge)$

- The *uniqueness quantifier* $\exists!$ meaning *"it exists a unique"*.
- The *exclusive quantifier* $\nexists$ meaning *"it doesn't exist"*.

It is also possible to change the nature of quantifiers by using a variable instead of restriction to retrieve a set of values (Hehner 2012):

$\S = f, x, ? \to \{\!|f(x) : ?|\!\}$

- The *solution quantifier* $\S$ meaning *"those"*.

It is interesting to note that most quantified expression can be expressed using the set builder notation discussed in the following section.

2.4 Set Theory

Since we need to represent knowledge, we will handle more complex data than simple booleans. One such way to describe more complex knowledge is by using set theory. It is used as the classical foundation of mathematics. Most other proposed foundations of mathematics invoke the concept of sets even before their first formula to describe the kind of notions they are introducing. The issue is then to define the sets themselves. At the beginning of his founding work on set theory, Cantor wrote:

Cantor (1895)

> "*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.*"

For Cantor, a set is a collection of concepts and percepts. In our case both notions are grouped in what we call *objects*, *entities* that are all ultimately *functions* in our formalism.

2.4.1 Base Definitions

This part is based on the work of Cantor (1895) and the set theory. The goal is to define the notions of set theory using our formalism.

Definition 12 (Set)**.** A collection of *distinct* objects considered as an object in its own right. We define a set one of two ways (always using braces):

- In extension by listing all the elements in the set: $\{0, 1, 2, 3, 4\}$
- In intention by specifying the rule that all elements follow: $\{n : ?(n)\}$

Using our functional foundation, we can define any set as a predicate $\mathcal{S} = e \to \top$ with e being a member of $\mathcal{S}$. This allows us to define the member function noted $e \in \mathcal{S}$ to indicate that e is an element of $\mathcal{S}$. $\in = e, \mathcal{S} \to \mathcal{S}(e)$

Another, useful definition using sets is the *domain* of a function f as the set of all arguments for which the function is defined. We call *co-domain* the domain of the inverse of a function. We can note them $f \ : \ \mathcal{D}(f) \mapsto \mathcal{D}(\bullet f)$. In the case of our functional version of sets, they are their own domain.

Definition 13 (Specification)**.** The *function of specification* (noted $:$) is a function that restricts the validity of an expression given a predicate. It can intuitively be read as *"such that"*. $(:) = f, ? \to f \triangledown (\mathcal{D}(? = \bot) \mapsto \mathcal{D}(\bullet f))$

The specification operator is extensively used in classical mathematics but informally, it is often seen as an extension of natural language and can be quite ambiguous. In the present document any usage of $(:)$ in any mathematical formula will follow the previously discussed definition.

2.4.2 Set Operations

Along with defining the domains of functions using sets, we can use function on sets. This is very important in order to define ZFC and is extensively used in the rest of the document.

In this section, basic set operations are presented. The first one is the subset.

Definition 14 (Subset)**.** A subset is a part of a set that is integrally contained within it. We note $\mathcal{S} \subset \mathcal{T} \vdash ((e \in \mathcal{S} \vdash e \in \mathcal{T}) \land \mathcal{S} \neq \mathcal{T})$, that a set $\mathcal{S}$ is a proper subset of a more general set $\mathcal{T}$. as

Definition 15 (Union)**.** The union of two or more sets $\mathcal{S}$ and $\mathcal{T}$ is the set that contains all elements in *either* set. We can note it:

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

Definition 16 (Intersection)**.** The intersection of two or more sets $\mathcal{S}$ and $\mathcal{T}$ is the set that contains only the elements member of *both* set. We can note it:

$$\mathcal{S} \cap \mathcal{T} = \{e \ : \ e \in \mathcal{S} \land e \in \mathcal{T}\}$$

Definition 17 (Difference)**.** The difference of one set $\mathcal{S}$ to another set $\mathcal{T}$ is the set that contains only the elements contained in the first but not the last. We can note it:

$$\mathcal{S} \setminus \mathcal{T} = \{e \ : \ e \in \mathcal{S} \land e \notin \mathcal{T}\}$$

An interesting way to visualize relationships with sets is by using Venn diagrams (Venn 1880). In figure 2.5 we present the classical union, intersection and difference operations. It also introduces a new way to represent more complicated notions such as the Cartesian product by using a representation for powerset and higher dimensionality inclusion that a 2D Venn diagram cannot represent.

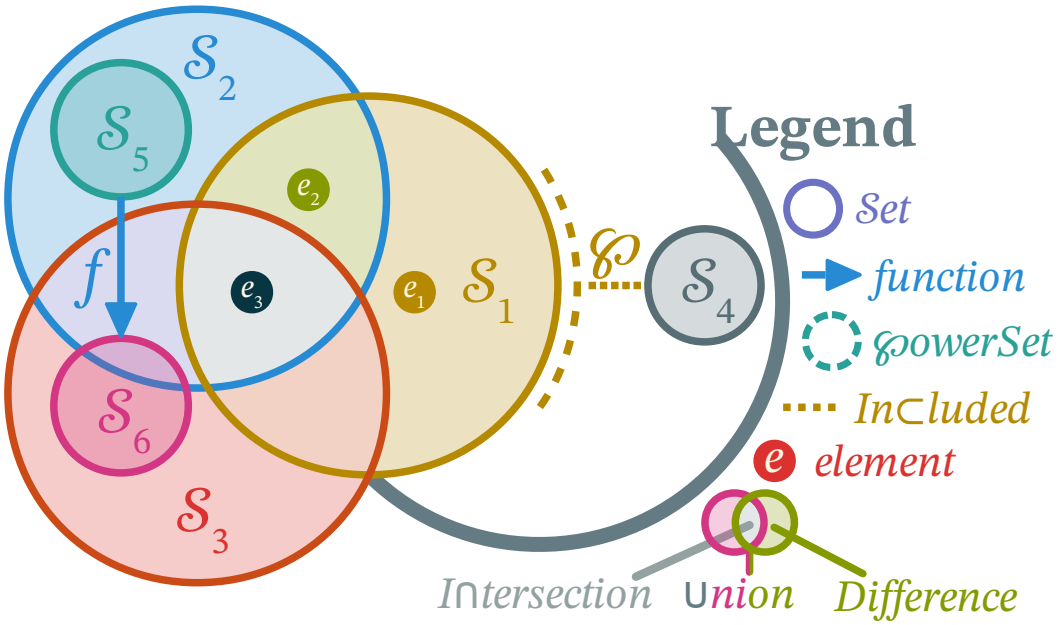


Figure 2.5: Example of an upgraded Venn diagram to illustrate operations on sets.

Example 10. Figure 2.5 is the graphical representation of the statements in table 2.3.

Table 2.3: Caption

Formula	Description
$e_1 \in \mathcal{S}_1$	e_1 is an element of the set $\mathcal{S}_1$.
$e_2 \in \mathcal{S}_1 \cap \mathcal{S}_2$	e_2 is an element of the intersection of $\mathcal{S}_1$ and $\mathcal{S}_2$.
$e_3 \in \mathcal{S}_1 \cap \mathcal{S}_2 \cap \mathcal{S}_3$	e_3 is an element of the intersection of $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}_3$.
$\mathcal{S}_5 \subset \mathcal{S}_2$	$\mathcal{S}_5$ is a subset of $\mathcal{S}_2$.
$\mathcal{S}_6 \subset \mathcal{S}_2 \cup \mathcal{S}_3$	$\mathcal{S}_6$ is a subset of the union of $\mathcal{S}_2$ and $\mathcal{S}_3$.
$f = \mathcal{S}_5 \mapsto \mathcal{S}_6$	f is a function which domain is $\mathcal{S}_5$ and co-domain is $\mathcal{S}_6$.
$\mathcal{S}_4 \subset \wp(\mathcal{S}_1)$	$\mathcal{S}_4$ is a combination of elements of $\mathcal{S}_1$.

These Venn diagrams, originally have a lack of expressivity regarding complex operations on sets. Indeed, from their dimensionality it is complicated to express numerous sets having intersection and disjunctions. For example, it is difficult to represent the following notion.

Definition 18 (Cartesian product)**.** The Cartesian product of two sets $\mathcal{S}$ and $\mathcal{T}$ is the set that contains all possible combinations of an element of both sets. These combinations are a kind of ordered set called *tuples*. We note this product:

$$\mathcal{S} \times \mathcal{T} = \{\langle e_\mathcal{S}, e_\mathcal{T} \rangle : e_\mathcal{S} \in \mathcal{S} \wedge e_\mathcal{T} \in \mathcal{T}\}$$

From this we can also define the set power recursively by $\mathcal{S}^1 = \mathcal{S}$ and $\mathcal{S}^n = \mathcal{S} \times \mathcal{S}^{n-1}$.

The Cartesian product can be seen as the set equivalent of currying. The angles $\langle\rangle$ notation is used for tuples, those are another view on currying by replacing several arguments using a single one as an ordered list. A tuple of two elements is called a *pair*, of three elements a *triple*, etc. We can access elements in tuples using their index in the following way $e_2 = \langle e_1, e_2, e_3 \rangle_2$. By decomposing the tuples as sets we can write:

$$\mathcal{S} \times \mathcal{T} = e_\mathcal{S}, e_\mathcal{T} \to \mathcal{S}(e_\mathcal{S}) \wedge \mathcal{T}(e_\mathcal{T})$$

Definition 19 (Mapping)**.** The mapping notation $\{\!|\}\!|$ is a function such that $\{\!|f(x) : x \in \mathcal{S}|\}\!|$ will give the result of applying all elements in set $\mathcal{S}$ as arguments of the function using the unCurrying operation recursively. If the function isn't specified, the mapping will select a member of the set non deterministically. The function isn't defined on empty sets or on sets with fewer members than arguments of the provided function.

Example 11. The classical sum operation on numbers can be noted:

$$\sum_{i=1}^{3} 2i = \{\!|+(2 * i) : i \in [1, 3]|\}\! = +(2 * 1)(+(2 * 2)(2 * 3))$$

2.4.3 The ZFC Theory

The most common axiomatic set theory is ZFC (Kunen 1980, vol. 102). In that definition of sets there are a few notions that come from its axioms. 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:

Axiom (Extensionality)**.** $\forall \mathcal{S} \forall \mathcal{T} : \forall e((e \in \mathcal{S}) = (e \in \mathcal{T})) \vdash \mathcal{S} = \mathcal{T}$

This means that two sets are equal if and only if they have all their members in common.

Another axiom of ZFC that is crucial in avoiding Russel's paradox ($\mathcal{S} \in \mathcal{S}$) is the following:

Axiom (Foundation)**.** $\forall \mathcal{S} : (\mathcal{S} \neq \varnothing \vdash \exists \mathcal{T} \in \mathcal{S}, (\mathcal{T} \cap \mathcal{S} = \varnothing))$

This axiom uses the empty set $\varnothing$ (also noted $\{\}$) as the set with no elements. Since two sets are equal if and only if they have precisely the same elements, the empty set is unique.

The definition by intention uses the set builder notation to define a set. It is composed of an expression and a predicate $?$ that will make any element e in a set $\mathcal{T}$ satisfying it part of the resulting set $\mathcal{S}$, or as formulated in ZFC: itS

Axiom (Specification)**.** $\forall ? \forall \mathcal{T} \exists \mathcal{S} : (\forall e \in \mathcal{S} : (e \in \mathcal{T} \wedge ?(e)))$

The last axiom of ZFC we use is to define the power set $\wp(\mathcal{S})$ as the set containing all subsets of a set $\mathcal{S}$:

Axiom (Power set). $\wp(\mathcal{S}) = \{\mathcal{T} : \mathcal{T} \subseteq \mathcal{S}\}$

With the symbol $\mathcal{S} \subseteq \mathcal{T} \vdash (\mathcal{S} \subset \mathcal{T} \vee \mathcal{S} = \mathcal{T})$. These symbols have an interesting property as they are often used as a partial order over sets.

2.5 Graphs

With set theory, it is possible to introduce all of standard mathematics. A field of interest for this thesis is the study of the structure of data. This interest arises from the need to encode semantic information in a knowledge base using a very simple language (see chapter 3). Most of these structures use graphs and isomorphic derivatives.

Definition 20 (Graph)**.** A graph is a mathematical structure g which is defined by its *connectivity function* χ that links two sets into a structure: the edges E and the vertices V.

2.5.1 Adjacency, Incidence and Connectivity

Definition 21 (Connectivity)**.** The connectivity function is a combination of the classical adjacency and incidence functions of the graph. It is defined using a circular definition in the following way:

Also: $\chi_{\multimap} = \bullet\chi_{\twoheadrightarrow}$

- *Adjacency*: $\chi_{\multimap} = v \to \{e : v \in \chi_{\twoheadrightarrow}(e)\}$
- *Incidence*: $\chi_{\twoheadrightarrow} = e \to \{v : e \in \chi_{\multimap}(v)\}$

Defining either function defines the graph. For convenience, the connectivity function combines the adjacency and incidence:

$$\chi = \chi_{\multimap} \bowtie \chi_{\twoheadrightarrow}$$

Usually, graphs are noted $g = (V, E)$ with the set of vertices V (also called nodes) and edges E (arcs) that links two vertices together. Each edge is classically a pair of vertices ordered or not depending on whether the graph is directed or not. It is possible to go from the set based definition to the functional relation using the following equation: $\mathcal{D}(\chi_{\twoheadrightarrow}) = E$

$E \subseteq V^2$

Example 12. A graph is often represented with lines or arrows linking points together like illustrated in figure 2.6. In that figure, the vertices v_1 and v_2 are connected through an undirected edge. Similarly v_3 connects to v_4 but not the opposite since they are bonded with a directed edge. The vertex v_8 is also connected to itself.

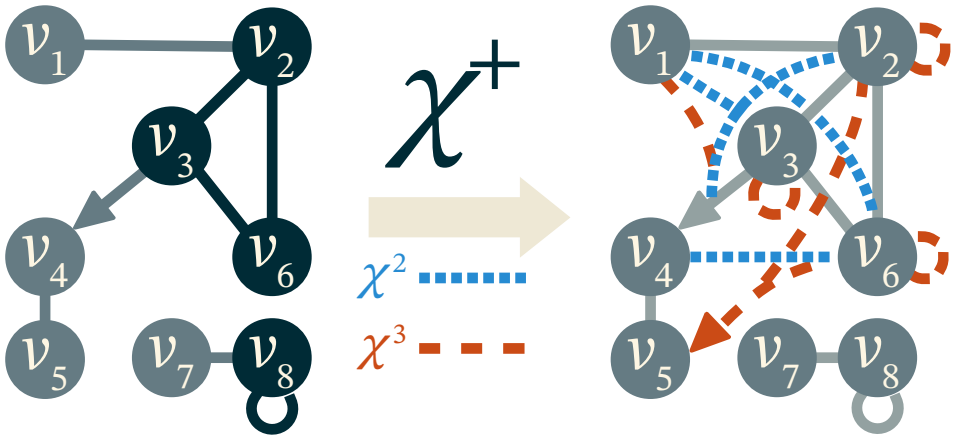


Figure 2.6: Example of the recursive application of the transitive cover to a graph.

2.5.2 Digraphs

The digraphs or *directional graphs* are a specific case of graphs where *all* edges have a direction. This means that we can have two vertices v_1 and v_2 linked by an edge and while it is possible to go from v_1 to v_2, the inverse is impossible. For such case the edges are ordered pairs and the incidence function can be decomposed into:

$$\chi_{\multimap} = \chi_{\succ\!\!\bullet} \bowtie \chi_{\bullet\!\!\prec}$$

We note $\chi_{\succ\!\!\bullet}$ the **incoming relation** and $\chi_{\bullet\!\!\prec}$ the **outgoing relation**.

In digraphs, classical edges can exist if allowed and will simply be bidirectional edges.

2.5.3 Path, cycles and transitivity

Most of the intrinsic information of a graph is contained within its structure. Exploring its properties requires to study the "shape" of a graph and to find relationships between vertices. That is why graph properties are easier to explain using the transitive cover χ^+ of any graph $g = (V, E)$.

This transitive cover will create another graph in which two vertices are connected through an edge if and only if it exists a path between them in the original graph g. We illustrate this process in figure 2.6. Note how there is no edge in $\chi^2(g)$ between v_5 and v_6 and the one in $\chi^3(g)$ is directed toward v_5 because there is no path back to v_6 since the edge between v_3 and v_4 is directed.

Definition 22 (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 $\langle v_1, v_2 \rangle \in \mathcal{D}(\chi^+(g))$.

The notion of connection can be extended to entire graphs. An undirected graph g is said to be *connected* if and only if $\forall e \in V^2(e \in \mathcal{D}(\chi^+(g)))$.

Similarly we define *cycles* as the existence of a path from a given vertex to itself. For example, in figure 2.6, the cycles of the original graph are colored in blue. Some graphs can be strictly acyclical, enforcing the absence of cycles.

2.5.4 Trees

A **tree** is a special case of a graph. A tree is an acyclical connected graph. If a special vertex called a *root* is chosen, we call the tree a *rooted tree*. It can then be a directed graph with all edges pointing away from the root. When progressing away from the root, we call the current vertex *parent* of all exterior *children* vertices. Vertex with no children are called *leaves* of the tree and the rest are called *branches*.

An interesting application of trees to FOL is called *and/or trees* where each vertex has two sets of children: one for conjunction and the other for disjunction. Each vertex is a logic formula and the leaves are atomic logic propositions. This is often used for logic problem reduction. In figure 2.7 we illustrate how and/or trees are often depicted.

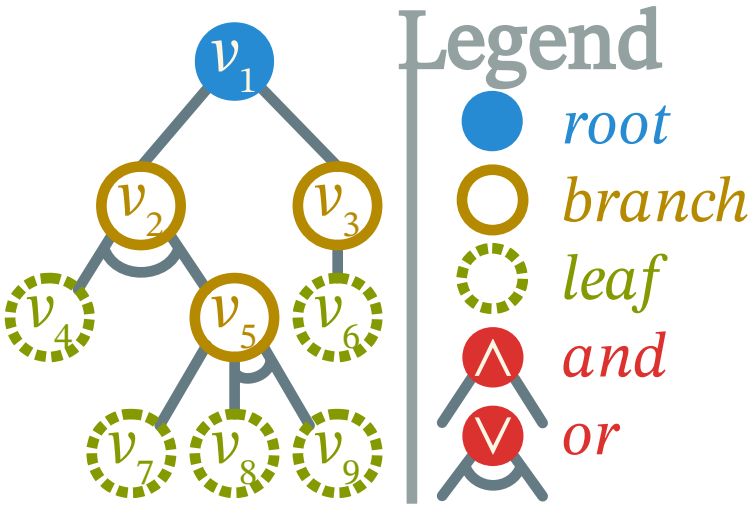


Figure 2.7: Example of and/or tree.

2.5.5 Quotient

Another notion often used for reducing big graphs is the quotiening as illustrated in figure 2.8.

Definition 23 (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. We note it $\div_f(g) =$

$(\{f(v) : v \in V\}, \{f(e) : e \in E\})$ with f being a function that maps any vertex either toward itself or toward its quotiened vertex.

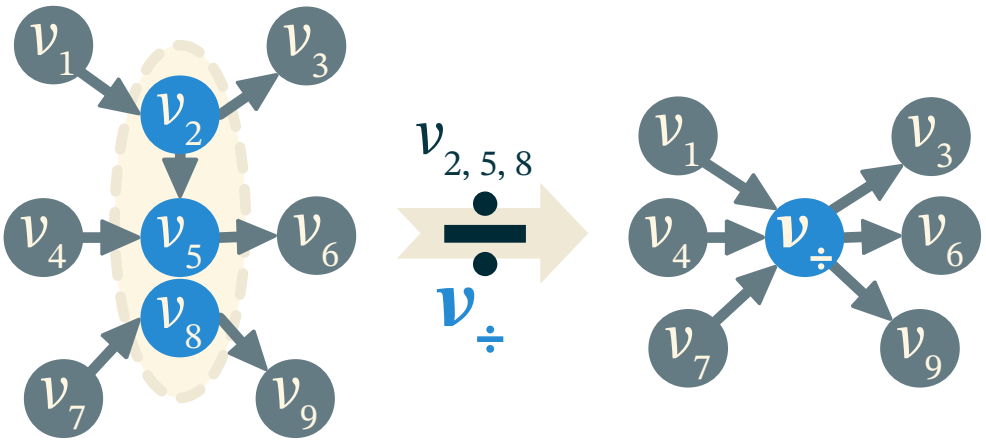


Figure 2.8: Example of graph quotient.

A quotient can be thought of as the operation of merging several vertices into one while keeping their connections with other vertices.

Example 13. Figure 2.8 explains how to do the quotient of a graph by merging the vertices v_2, v_5 and v_8 into $v_\div$. The edge between v_2 and v_5 is lost since it is inside the quotienned part of the graph. All other edges are now connected to the new vertex $v_\div$.

2.5.6 Hypergraphs

A generalization of graphs are **hypergraphs** where the edges are allowed to connect to more than two vertices (Ray-Chaudhuri and Berge 1972). They are often represented using Venn-like representations but can also be represented with edges "gluing" several vertex like in figure 2.9.

Example 14. In figure 2.9, vertices are the discs and edges are either lines or gluing surfaces. In hypergraph, classical edges can exist like e_4, e_6 or e_7. Taking for example e_1, we can see that it connects 3 vertices: v_1, v_2 and v_3. It is also possible to have an edge connecting edges like e_8 that connects e_3 to itself. Edges can also "glue" more than two edges like e_2 connects e_1, e_3 and e_4. The most exotic structures are edge-loops as seen with e_9 and e_{10} which allow graphs that are only made of edges without any vertices.

An hypergraph is said to be *n-uniform* if the edges are restricted to connect to only n vertices together. In that regard, classical graphs are 2-uniform hypergraphs.

Hypergraphs have a special case where $E \subset V$. This means that edges are allowed to connect to other edges. In figure 2.9, this is illustrated by the edge e_2 connecting

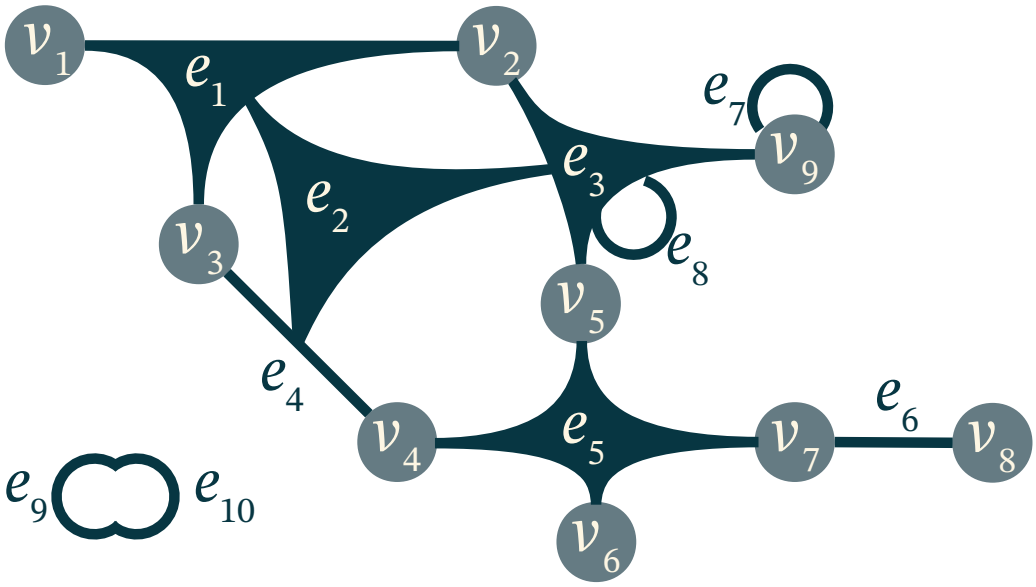


Figure 2.9: Example of hypergraph with total freedom on the edges specification.

to three other edges. That type of edge-graphs are akin to port graphs (Silberschatz 1981). An interesting discussion about the compatibility of hypergraphs with ZFC is said presented by Vepstas (2008). He says that a generalization of hypergraph allowing for edge-to-edge connections violate the axiom of Foundation of ZFC by allowing edge loops. Indeed, like in figure 2.9, an edge $e_9 = \{e_{10}\}$ can connect to another edge $e_{10} = \{e_9\}$ causing an infinite descent inside the $\in$ relation in direct contradiction with ZFC.

This shows the limitations of FOL and ZFC-based models, particularly in the field of knowledge representation. Some structures require higher dimensions as proposed by HOL, modal logic and hypergraphs. However, it is important to note that these models are more general than those based on FOL and ZFC. Indeed, these models contain what is possible to represent in a classical way but remove restrictions specific to these models.

2.6 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 24 (Seed)**.** A seed corresponds to a vertex along with the set of adjacent builT edges. Formally we note a seed $\star = (v, \chi_g(v))$ that means that a seed build from the vertex v in the graph g contains a set of adjacent edges $\chi_g(v)$. We call the vertex v the *germ* of the seed. All edges in a seed do not connect to the other vertices but keep the information and are able to match the correct vertices through typing (often a type of a single individual). We call the edges in a seed *connectors*.

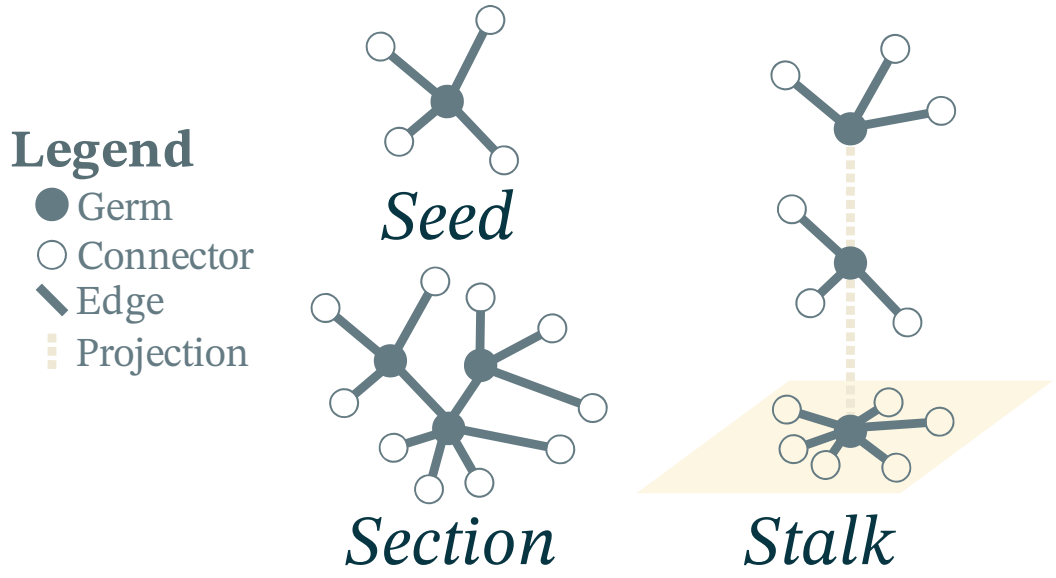


Figure 2.10: Example of a seed, a section and a stalk.

Seeds are extracts of graphs that contain all information about a vertex. Illustrated in the figure 2.10, seeds have a central germ (represented with discs) and connectors leading to a typed vertex (outlined circles). Those external vertices are not directly contained in the seed but the information about what vertex can fit in them is kept. It is useful to represent connectors like jigsaw puzzle pieces: they can match only a restricted number of other pieces that match their shape.

From there, it is useful to build a kind of partial graph from seeds called sections.

Definition 25 (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. We note $g_\star = (V, \{\!\{\cup : E_{section}\}\!\})$ the graph formed by the section.

In figure 2.10, a section is represented. It is a connected section composed of seeds along with the additional seeds of any vertices they have in common. They are very similar to subgraph but with an additional border of typed connectors. This tool was originally mostly meant for big data and categorization over large graphs. As the graph quotient is often used in that domain, it was transposed to sections. Quotients allow us to define stalks.

Definition 26 (Stalk)**.** Given a projection function $f : V \to V'$ over the germs of a section $\star$, the stalk above the vertex $v' \in V'$ is the quotient of all seeds that have their germ follow $f(v) = v'$.

The quotienning is used in stalks for their projection. Indeed, as shown in figure 2.10, the stalks are simply a collection of seeds with their germs quotiened into their common projection. The projection can be any process of transformation getting a set of

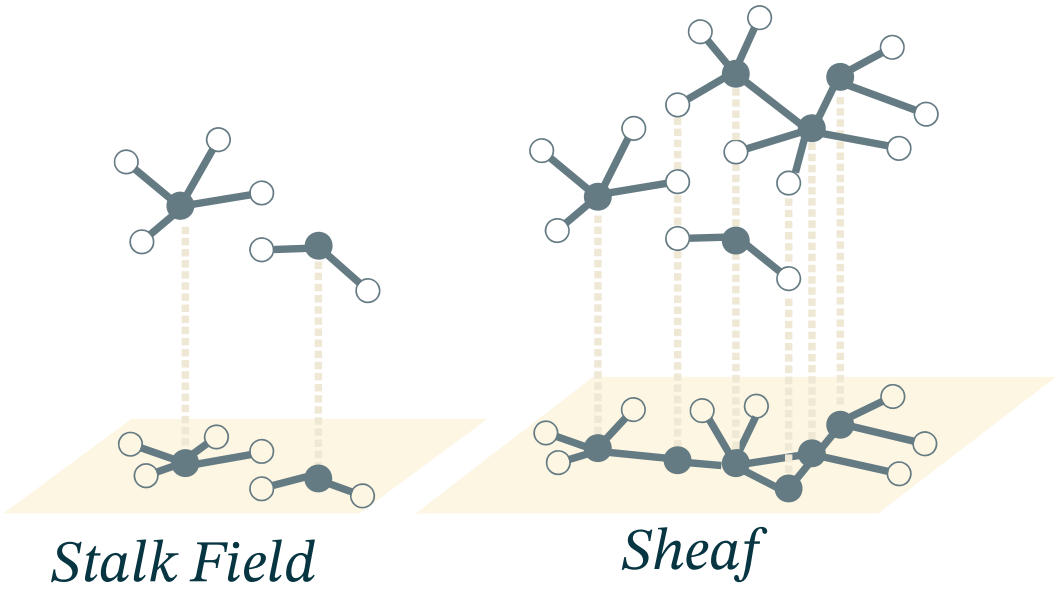


Figure 2.11: Example of sheaves.

seeds in one side and gives object in any base space called the image. Sheaves are a generalization of this concept to sections.

Definition 27 (Sheaf)**.** A sheaf is a collection of sections, together with a projection. We note it $\mathcal{F} = \langle g_\star, glue \rangle$ with the function $glue$ being the gluing axioms that the projection should respect depending on the application. The projected sheaf graph is noted as the fusion of all quotiened sections:

$$glue_{\mathcal{F}} = \{\div_{glue_\star} : \{glue_\star \in g_\star\}$$

By putting several sections into one projection, we can build stack fields. These fields are simply a subcategory of sheaves. Illustrated in figure 2.11, a sheaf is a set of sections with a projection relation that usually merges similarly typed connectors.

2.7 Conclusion

In this chapter, we presented the tools we will use for the rest of the document along with its underlying formalism. First we presented a functional theory that allows for a concise expression of the formula for our usage. We also described classical mathematical tools like FOL, set theory and graphs. In parallel, we introduced non-classical tools of higher order such as HOL, modal logic, hypergraphs and sheaves. Those notions are mostly data structures and allow to express any model needed for our usage.

The first of these models is about a partial self described language for knowledge representation.

3 Knowledge Representation

Figure 3.1: Noam Chomsky 2017

Knowledge representation is at the intersection of maths, logic, language and computer sciences. Knowledge description systems rely on syntax to interoperate systems and users to one another. The base of such languages comes from the formalization of automated grammars by Chomsky (1956). It mostly consists of a set of production rules aiming to describe all accepted input strings. Usually, the rules are hierarchical and deconstruct the input using simpler rules until it matches a terminal symbol. This deconstruction is called parsing and is a common operation in computer science. More tools for the characterization 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) metalanguage was created on top of Chomsky's formalization.

A similar process happened in the 1970s, when logic based knowledge representation gained popularity among computer scientists (Baader *et al.* 2003). Systems at the time explored notions such as rules and networks to try and organize knowledge into a rigorous structure. At the same time other systems were built based on First Order Logic (FOL). Then, around the 1990s, the research began to merge in search of common semantics in what led to the development 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, actors of the internet were on the lookout for standardization and interoperability of computer systems. One such standardization took the name of "semantic web" and aimed to create a widespread network of connected services sharing knowledge between one another in a common language. At the beginning of the 21st century, several languages were created, all based on the World Wide Web Consortium (W3C) specifications called Resource Description Framework (RDF) (Klyne and Carroll 2004). This language is based on the notion of statements as triples. Each can express a unit of knowledge. All the underlying theoretical work of DL continued with it and created more expressive derivatives. One such derivative is the family of languages called Web Ontology Language (OWL) (Horrocks *et al.* 2003). The 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.

Nowadays, when designing a knowledge representation, one usually starts with existing framework. The most popular in practice is certainly the classical relational database, followed closely by more novel methods for either big data or more expressive solutions like ontologies. Even if they are not strickly isomorphic to ontologies, relational databases are often used in lieu of it and some works even aim to convert ontologies into databases while keeping their informations (Bienvenu 2018).

In this chapter, we present a new tool that is more expressive than ontologies while remaining efficient. This model is based on dynamic grammar and basically is defined mostly by the structure of knowledge. Our model is inspired from RDF triplets, especially in its Turtle syntax (W3C 2014). Of course, this will lead to compromises, but can also have some interesting properties. This knowledge representation system will allow us to express hierarchical planning domains in chapter 6.

3.1 Grammar and Parsing

Grammar is an old tool that used to be dedicated to linguists. With the funding works by Chomsky and his Context-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.

3.1.1 BNF

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

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

A terminal symbol is a rule that does not depend on 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 forms.

In this section, we introduce 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 preceding expression. We also add the negation operator `~` that matches only if the following expression does not match. We also add parentheses for grouping expression and brackets to group literals.

Example 15. We can make a grammar for all sequence of `A` using the rule `<scream> ::= "A"+`. If we want to make a rule that prevent the use of the letter `z` we can write `<no-sleep> ::= ~"z"`.

3.1.2 Tools for text analysis

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 and explain our choice of grammatical framework.

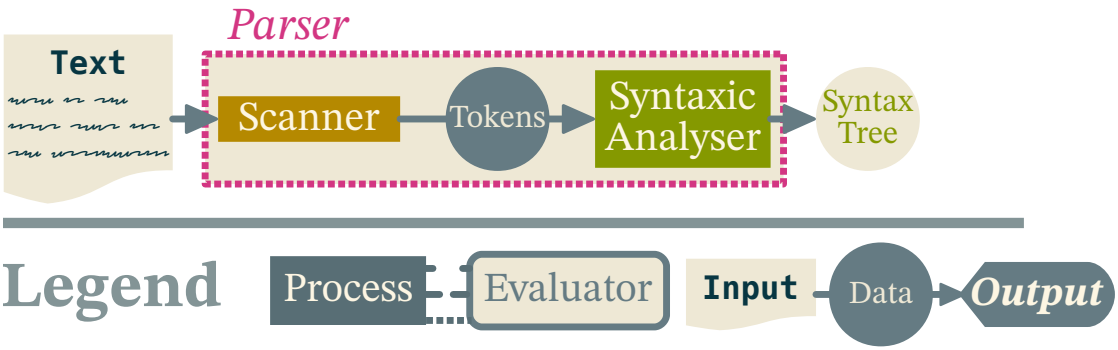


Figure 3.2: Process of a parser while analyzing text

One of the main tools for both static and dynamic grammar is a parser. It is the program that will decode the input into a *syntax tree*. This process is detailed in figure 3.2. To do that it first scans the input text for matching *tokens*. Tokens are akin to words and are the data unit at the lexical level. Then the tokens are matched against production rules of the parser (usually in the form of a grammar). The matching of a rule will add an entry into the resulting tree that is akin to the hierarchical grammatical description of a sentence (e.g. proposition, complement, verb subject, etc.). manque virgule

Most of the time, a parser will be used with an *evaluator*. This component transforms the syntax tree into another similarly structured result. 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 compiler simpler. We call those tools compiler-compilers or parser generators (Paulson 1982). They take a grammar description as input and gives the program of a compiler of the generated language as an output. Figure 3.3 explains how both the generation and resulting program work. Each of them uses a parser linked to an *evaluator*. In the case of a compiler-compiler, the evaluator is actually a compiler process. It will transform the syntax tree of the grammar into executable code. This code is the generated compiler and is subject to our interest in this case.

3.1.3 Dynamic Grammar

One of the issues with classical grammars is that they are set in stone once compiled. One cannot change the definition of a grammar without editing the grammar definition and compiling it. Although this might be more than sufficient for most usages, it can hinder the adaptability of a general-purpose tool. In this section we present the existing types of dynamic grammar and their advantages and limitations.

For dynamic grammar, compilers can get more complicated. The most straightforward way to make a parser able to handle a dynamic grammar 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 is grammar that can modify themselves. In order to do this a grammar is valuated with reified objects representing parts of itself (Hut-

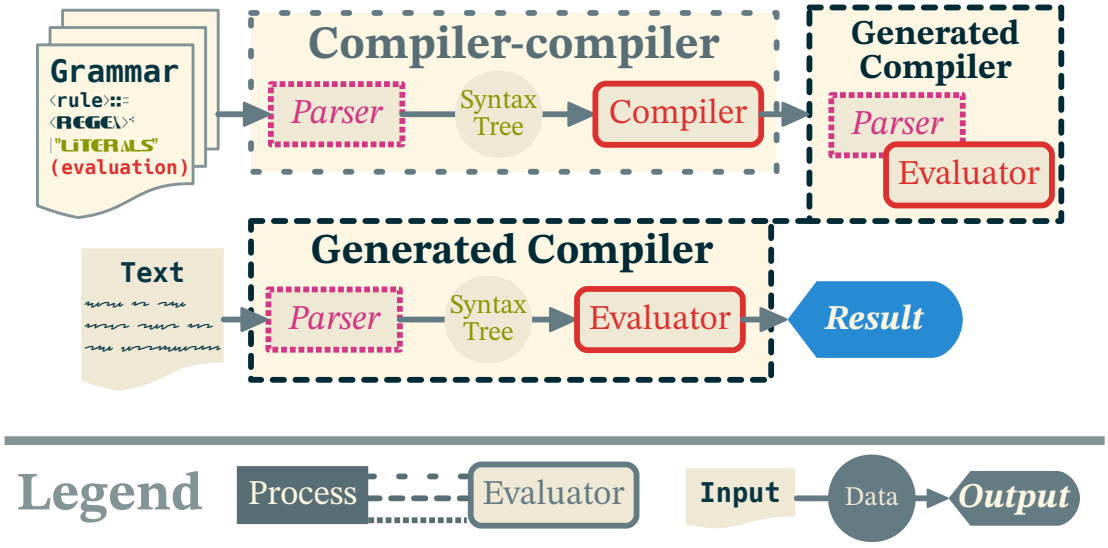


Figure 3.3: Illustration of the meta-process of compiler generation

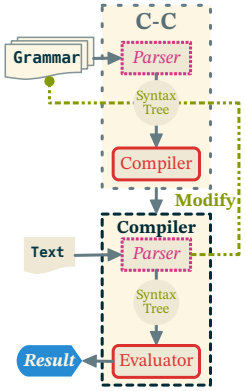


Figure 3.4: Illustration of the dynamic grammar modification

ton and Meijer 1996). These parts can be modified dynamically by rules as the input gets parsed (Renggli *et al.* 2010; Alessandro and Piumarta 2007). Reusing our prior illustration, we can show in figure 3.4, the particularity of this type of grammar. This approach uses Parsing Expression Grammars (PEG)(Ford 2004) with Packrat parsing that 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 is easier to implement and more efficient in practice than their classical counterparts (Loff *et al.* 2018; Henglein and Rasmussen 2017), it offsets 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 software (Höschele and Zeller 2017; Grünwald 1996). However, these kinds of approaches require a lot of input data to perform well. They also simply provide the grammar after expensive computations.

My system uses a grammar, composed of classical rules and is extended using meta-rules that activate once the classical grammar fails.

3.2 Description Logics

One

On of the most standard and flexible way of representing knowledge is by using ontologies. They are based mostly on the formalism of Description Logics (DL) (Krötzsch *et al.* 2013). It is based on the notion of classes (or types) as a way to make the knowledge hierarchically structured. A class is a set of individuals that are called instances of

have

the classes. Classes got the same basic properties as sets but can also be constrained with logic formula. Constraints can be on anything about the class or its individuals. Knowledge is also encoded in relations that are predicates over attributes of individuals.

It is common when using DLs to store statements into three boxes (Baader *et al.* 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 entity behavior. It restricts usages of roles (properties) in the ABox. The terminology is often hierarchically ordered using a subsumption relation noted $\subseteq$. If we represent classes or type as a set of individuals then this relation is akin to the subset relation of set theory.

Example 16. In the classical genealogy example (Baader *et al.* 2003), the TBox can be a statement similar to $\text{Woman} = \text{Person} \cap \text{Female}$. This is reasoning about the concept hierarchy and is usually modeled at design time.

The RBox is often not present in DL systems but have interesting expressivity properties. For example, it is possible to define the atomic role gender so that $\text{Person} \cap \forall \text{gender} \in \{\text{Male}, \text{Female}, \text{NonBinary}\}$. This will enforce that every person should have one of the three genders exposed in the set.

The ABox is about instances like $\text{Female} \cup \text{Person}(\text{ALICE})$ stating that Alice is a female and a person. This statement allows the system to infer that $\text{Woman}(\text{ALICE})$ by applying the rules of the TBox and RBox.

There are several versions and extensions of DL. They all vary in expressivity. Improving the expressivity of a DL system often comes at the cost of less efficient inference engines that can even become undecidable for some extensions of DL.

3.3 Ontologies and their Languages

Most AI problem needs a way to represent knowledge. The classical way to do so has been more and more specialized for each AI community. Every domain uses its Domain Specific Language (DSL) that neatly fits 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". This domain has given numerous works on service composition that is very close to hierarchical planning (Rao *et al.* 2004).

From numerous works, a repeated limitation of the "semantic web" seems to come from the languages used (Dornhege *et al.* 2012; Hirankitti and Xuan 2011). 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.

The most basic of these languages is perhaps RDF Turtle (Beckett and Berners-Lee 2011). It is based on triples 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 (V, E) with V being the set of RDF nodes and E being the set of edges. This graph also comes with an incomplete label relation that associates a unique string called a Uniform Resource Identifier (URI) to most nodes. 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.

Example 17. To illustrate how RDF is used, we present in listing 3.1, an example from the W3C (2004a). This example is from the famous "Library" example commonly used to explain relational databases. The short triple representation in that listing is possible thanks to the Turtle variant of RDF (Beckett and Berners-Lee 2011). This example also shows the use for properties from the `rdf:` namespace. These properties are fundamental to RDF and allow standard description of basic properties such as the type.

```
1   ex:ontology rdf:type owl:Ontology .
2   ex:name rdf:type owl:DatatypeProperty .
3   ex:author rdf:type owl:ObjectProperty .
4   ex:Book rdf:type owl:Class .
5   ex:Person rdf:type owl:Class .
6
7   _:x rdf:type ex:Book .
8   _:x ex:author _:x1 .
9   _:x1 rdf:type ex:Person .
10  _:x1 ex:name "Fred"^^xsd:string .
```

Listing 3.1: Example of RDF turtle ontology

Built on top of RDF, the W3C recommended another standard called OWL (W3C 2012). It adds the ability to have hierarchical classes and properties along with more advanced description of their arrity and constraints. OWL is, in a way, more expressive than RDF (Van Harmelen *et al.* 2008, 1,p825). OWL comes in three versions: OWL Lite, OWL DL and OWL Full. The lite version is less advanced but its inference is decidable, OWL DL contains all notions of DL and the full version contains all features of OWL but is strongly undecidable.

The expressivity can also come from a lack of restriction. If we allow some freedom of expression in RDF statements, its inference can quickly become undecidable (Motik 2007). This kind of extremely permissive language is better suited for specific usage for other branches of AI. Even with this expressivity, several works still deem existing ontology system 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 used for information about any statement represented as a triple (or 3 fields as the subject property and object). These include context, annotations, access rights, probabilities, or most of the time the source of the data (Tolksdorf *et al.* 2004). One of the other uses of the fourth field of RDF is to reify statements (Hernández *et al.* 2015). The reification is a compound process. It needs two steps:

- *abstraction* or generalization of the relational structure of a concept. It can be seen as an imperfect description of an object.
- *symbolization* or referring to another structure as being equivalent to a single object. It can be seen as "compressing" the information into one symbol.

In RDF, reification is the act of describing a statement using special relations such as `rdf:subject`, `rdf:property` and `rdf:object`. Then the node describing the statement is typed as a statement and can be used in high order knowledge. Consequently, by identifying each statement, it becomes possible to efficiently form statements about any statements.

Reifying isn't the only way to express reflexivity in ontologies. In the work of Toro *et al.* (2008), the solution explored is to encode queries into the ontology. This allows for query caching and certainly adds to the expressivity. However, encoding queries will only be relevant once queries are already executed.

3.4 Limits

The issue with using these classical tools is that they are very hard to combine. Indeed, making ontologies with a dynamic grammar is out of the question when using the main ontology frameworks. This difficulty is only slightly alleviated when trying to build an ontology framework on top of a dynamic grammar. This lack of adaptability or expressivity is the reason why other approaches must be considered.

Hart and Goertzel (2008) uses a different approach in their framework for Artificial General Intelligence (AGI) called OpenCog. The structure of the knowledge is based on a rhizome, a collection of trees, linked to each other. This structure is called Atomspace. Each vertex in the tree is an atom, leaf vertices are nodes, the others are links. Atoms are immutable, indexed objects. ~~They can be~~ given ~~that~~ values can be dynamic 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.

In this chapter, we present a similar knowledge structure as AtomSpace that is used along with notions inspired by ontology. The next section presents our contribution toward a knowledge description framework that allows native higher order representation needed for hierarchical planning.

3.5 Structurally Expressive Language Framework (SELF)

As we have seen, the most used knowledge description systems (e.g. RDF, ontologies and relational databases) have a common drawback: they are static. This means that they are created to be optimized for a specific use cases, ~~or~~ gets general at the cost of efficiency.

This issue is mainly due to the lack of flexibility of the language. Since the grammars used for ontologies are static, the language cannot be modified unless manually and by recompiling the tools.

The main issue is that such systems are unable to adapt to the use case by themselves. To fix this issue, a new knowledge representation model must be presented. We propose to base our framework on dynamic grammar and exploit the properties of the grammar to make the knowledge description evolve to fit its usage.

The goal is to make a minimal language framework that can adapt to its use to become as specific as needed. If it becomes specific, it 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 framework must maximize the following criteria:

- **Neutral**: Must be independent from preferences and regional localization.
- **Permissive**: Must allow as many data representation as possible.
- **Minimalist**: Must have the minimum number of base axioms and as little native notions as possible.
- **Adaptive**: Must be able to react to user input and be as flexible as possible.

Table 3.1: Comparison of different approaches using our criteria.

Approaches	Neutral	Permissive	Minimalist	Adaptative
Relational	- -	- - -	- -	-
Triple	+	++	+	+
Ontology	-	+	-	-
AtomSpace	++	+++	- -	++
SELF	+++	+++	++	+++

Table 3.1 presents the fitness of each approach for each criterion. The first approach is the relational database. While widely used, this approach requires an extensive definition of the database schema and, while using simple syntax, is quite verbose in comparison of modern languages. The second approach is the triple representation of RDF. While it allows for more possibilities of expression, it still requires some specific node URIs in order to express higher order knowledge. The ontologies don't have many advantages. They can be more expressive but the added restrictions on interpretation makes it less appealing for our use. Indeed, it needs library worth of specific core URIs and its typing system restricts the ability to abstract even more, especially on OWL Lite. What it gains in speed and desirability, it loses on flexibility. The last existing approach is the AtomSpace of OpenCog. This knowledge base allows for very abstract structures. It, however, is quite heavy and not meant to be directly understood by human readers. This along with the time needed for inference makes it an unfit approach for our objectives.

In order to respect these requirements, we developed a framework for knowledge description. This Structurally Expressive Language Framework (SELF) is our answer to these criteria. SELF is inspired by RDF Turtle and Description Logic.

3.5.1 Knowledge Structure

SELF 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 SELF graph is a set of statements that transparently include their own identity. The closest representation of the underlying structure of SELF is as follows:

$$g_{\mathbb{U}} = (\mathbb{U}, S) : S = \{s = \langle sub, pro, obj \rangle : s \in \mathcal{D} \vdash s \wedge \mathcal{D}\}$$

with:

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

This means that the world $g_{\mathbb{U}}$ is a graph with the set of entities $\mathbb{U}$ as vertices and the set of statements S as edges. This model also supposes that every statement s must be true if it belongs to the domain $\mathcal{D}$. This graph is a directed 3-uniform hypergraph.

See section 2.5.6.

Since sheaves are a representation of hypergraphs, we can encode the structure of SELF into a sheaf-like form. Each seed is a statement, the germ being the statement vertex. It is always accompanied by an incoming connector (its subject), an outgoing connector (its object) and a non-directed connector (its property). The sections are domains and must be coherent. Each statement, along with its property, makes a stalk as illustrated in figure 3.5.

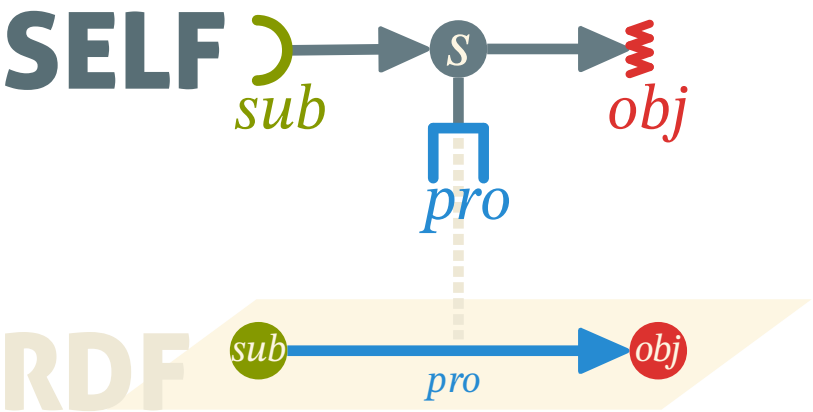


Figure 3.5: Projection of a statement from the SELF to RDF space.

The difference with a sheaf is that the projection function is able to map the pair statement-property into a labeled edge in its projection space. We map this pair into

a classical labeled edge that connects the subject to the object of the statement in a directed fashion. This results in the projected structure being a correct RDF graph.

3.5.1.1 Consequences

The base knowledge structure is more than simply convenience. The fact that 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 are related to information about the subject, property and object of a designated statement. The problem is that such statements require three regular statements just to be defined. Using the fourth field, it becomes possible to make statements about *any* statements. It also becomes possible to express modal logic about statements or to express, various traits like the probability or the access rights of a statement.

pas e virgule

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 SELF. The axiom of Structure is the only axiom of the system. From this axiom it is possible to derive theorems that are logical propositions ~~that are~~ directly deductible from the axioms of a system.

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

This theorem comes from the axiom of Extensionality of ZFC. 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. This notion of identity entails that a given entity cannot change in a way that would alter its identifier.

aN

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 system and is often a requirement of such constructs. This theorem is also a consequence of the axiom of Structure: $s \in \mathcal{D} \vdash s \wedge \mathcal{D}$.

Theorem 3 (Uniformity)**.** *Any object in SELF is an entity. Any relations in SELF are restricted to* $\mathbb{U}$*.*

This also means that all native relations are closed under $\mathbb{U}$. This allows for a uniform knowledge database.

3.5.1.2 Native Properties

In the following, we suppose all notions from previous chapter. The difference is that we define and use only a subset of the functions defined in the SELF formalism. In relation to the theory of SELF, we use the functional theory previously defined as the underlying formalism.

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

The **equality relation** $=: \mathbb{U} \mapsto \mathbb{U}$, behaves like the classical operator. Since the knowledge database will be expressed through text, we also need to add an explicit way to identify entities. This identification is done through the **name relation** $\nu : \mathbb{U} \mapsto L_{String}$ that affects a string literal to some entities. This leads us to introduce literals into SELF that are also entities that have a native value.

The axiom of Structure puts a type restriction on property. Since it compartments $\mathbb{U}$ using various named subsets, we must adequately introduce an explicit type system into SELF. That type system requires a **type relation** named using the colon (:). It is noted $(:) : \mathbb{U} \mapsto T$. 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), introduces a **subsumption relation** $\subseteq: T \mapsto T$ as a partial ordering relation to the types. Since types can be seen as sets of instances, we simply use the subset relation from set theory. In our case, the entity type is the greatest element of the lattice formed by the set of types with the subsumption relation $(T, \subseteq)$.

The theorem 3 also allows for some very interesting meta-constructs. That is why we also introduce a signed **Meta relation** $\mu : \mathbb{U} \mapsto D$ with $\mu^\bullet = \bullet\mu$. This allows to create domain from certain entities and to encapsulate domains into entities. $\mu^\bullet$ is for reification and μ is for abstraction. This Meta relation also allows to express value of entities, like lists or various containers.

To fulfill the principle of adaptability and in order to make the type system more useful, we introduce the **parameter relation** $\rho : \mathbb{U} \mapsto \mathbb{U}$. This relation affects a list of parameters, using the Meta relation, to some parameterized entities. This also allows for variables in statements.

Since axiom of Structure gives the structure of SELF a hypergraph shape, we must port some notions of graph theory into our framework. Introducing the **statement relation** $\chi : S \mapsto \mathbb{U}$ reusing the same symbol as for the adjacency and incidence relation of graphs. This isn't a coincidence as this relation has the same properties.

ce n'est pas une phrase il manque le verbe ?

Example 18. Since statements are triplets and edges, s_0 gives the subject of a statement s. Respectively, s_1 and s_2 give the property and object of any statement. For adjacencies, χ can give the set of statements any entity is the object or subject of. For any property pro, the notation $\chi(pro)$ gives the set of statements using this property.

These definitions allow us to build the hypergraph structure by using basic graph formalism.

All of this structure along with the native relations are presented in table 3.2. Figure 3.6 illustrates the way those sets and relations interact with one another. The Venn diagram of SELF is contained within $\mathbb{U}$ since it is endomorphic. It is interesting to notice that the domains ($\mathcal{D}$) is a subset of the powerset of statements (S.

Table 3.2: List of symbols of the SELF formalism.

Set	Description	Relation	Description
$\mathbb{U}$	Entities	$=$	Equality

Set	Description	Relation	Description
P	Properties	ρ	Parameter
S	Statements	χ	Statement
$\mathcal{D}$	Domain	μ	Meta
T	Types	:	Type
L	Literals	$\subseteq$	Subsumption
$String$	Strings	ν	Name

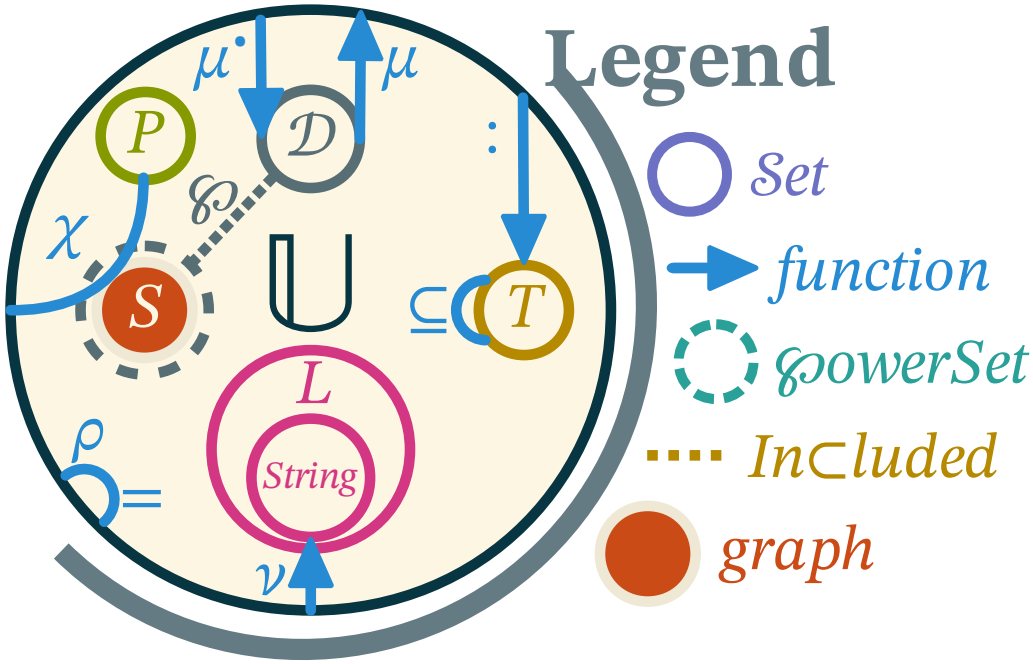


Figure 3.6: Venn diagram of subsets of $\mathbb{U}$ along with their relations. Dotted lines mean that the sets are defined a subset of the wider set.

3.5.2 Syntax

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 so verbose that it is not used for most examples in the documents that defines RDF because it is too confusing and complex (W3C 2004b; W3C 2004c). The XML syntax is also quite restrictive and cannot evolve dynamically to adapt to the usage.

The problem is that our principles can be contradictive. Indeed, a general language that is very permissive is often far from minimalist or adaptive. A potential solution would be to use two languages:

- A first one that is general and minimalistic.

- A second one that is permissive and adaptive.

The issue is that we certainly don't want users to have to learn two separate languages to use our framework. Also the second language would be complicated to describe since it will be specific and will need to fit any particular use cases.

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

3.5.2.1 Grammar $\mathbb{g}_0$

The description of $\mathbb{g}_0$ 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 3.2, we give a simplified version of $\mathbb{g}_0$. It is written in a pseudo-BNF fashion, which 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 separators for the other rules.
- `<?name>` are inferred rules and tokens. They play a key role for the process of derivation explained in section 3.5.3.

```
1 <~COMMENT: <INLINE: "//" (~["\n", "\r"])*>
2 | <BLOCK: "/*" (~["*/"])*> > //Ignored
3 <~WHITE_SPACE: " "|"\t"|"\n"|"\r"|"\f">
4 <LITERAL: <INT> | <FLOAT> | <CHAR> | <STRING>> //Java definition
5 <ID: <TYPE: <UPPERCASE>(<LETTERS>|<DIGITS>)* >
6 | <ENTITY:  <LOWERCASE>(<LETTERS>|<DIGITS>)*>
7 | <SYMBOL: (~[<LITERALS>, <LETTERS>, <DIGITS>])*>>
8
9 <worselfld> ::= <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 3.2: Simplified pseudo-BNF description for basic SELF.

In $\mathbb{g}_0$, the first two token definitions are ignored. This means that comments and whitespaces 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.

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

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 $\mu^\bullet$). The most useful token

has been ? où c'est ton numéro de section qui est mauvais

`<EOS>` is used as a statement delimiter. This rule also permits the inclusion of other files if a string literal is used as a subject. The underlying logic of this first statement will be presented in section 3.5.5.1. In the following examples we will consider that `<EQUAL> ::= "="`, `<SOLVE> ::= "?"` and `<EOS> ::= ";"`.

At line 11, we can see one of the most defining features of $\mathbb{g}_0$: statements. The input is nothing but a set of statements. Each component of the statements are entities. We defined two specific rules for the subject and object to allow for eventual runtime modifications. The property rule is more restricted in order to guarantee the non-ambiguity of the grammar.

is

entiTY

3.5.2.2 Neutrality and encoding

In order to respect the principle of neutrality, the language must not suppose any regional predisposition of the user. There are few exceptions 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 occidental 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 concession is made for reasons of simplicity and efficiency, but it is possible to define literals dynamically in theory (see section 8.1.1.1).

The principle of neutrality makes mandatory to use an extensive character encoding standard in order to support non-roman languages. The best candidate for such an encoding is the Unicode Transformation Format – 8-bit (UTF-8 see Unicode Consortium 2018a, chap. 2). This standard comes with a database that names, categorize and describe all characters. That database is called Unicode Character Database.

S

describeS

UCD Unicode Consortium (2018b)

Even if sticking to the ASCII subset of characters is a good idea for efficiency, SELF can work with UTF-8 and exploits the UCD for its token definitions. This means that SELF comes keywords 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 which is closer to an imposed configuration file).

White-spaces are defined against UCD's definition of the separator category `Z&` (see Unicode Consortium 2018a, chap. 4).

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 categories `Lu`, `Ll`, `L&`, `Nd`). This means that any language's uppercase can be used in that context. For performance and simplicity reasons we will only use ASCII in our examples and application.

3.5.3 Dynamic Grammar

The syntax we described is only valid for $\mathbb{g}_0$. As long as the input is conforming to these rules, the framework keeps the minimal behavior. In order to access more features, one needs to break a rule. We add a second outcome to handling with violations :

handle

derivation. There are several kinds 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 cause 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 $\mathbb{g}_0$: 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 reducible to a triple and to augment $\mathbb{g}_0$ by adding an expression to any `<meta_*>` rules. If the input has fewer than 3 entities 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.

3.5.3.1 Containers

The first meta-rule is to infer a container. A container is delimited by, at least, a left and a right delimiter (they can be the same symbol). An optional middle delimiter can also be used but must be different from any other delimiters. We infer the delimiters using the algorithm 1.

Algorithm 1 Container meta-rule

```
1:  function CONTAINER(Token current)
2:      LOOKAHEAD(current, EOS)                          ▷ Populate all tokens of the statement
3:      for all token in horizon do
4:          if token is a new symbol then delimiters.APPEND(token)
5:      if LENGTH(delimiters) <2 then
6:          if COHERENTDELIMITERS(horizon, delimiters[0]) then
7:              INFERMIDDLE(delimiters[0])               ▷ New middle delimiter in existing containers
8:              return Success
9:          return Failure
10:     while LENGTH(delimiters) > 0 do
11:         for all (left, middle, right) in SORTEDDELIMITERS(delimiters) do
12:             if COHERENTDELIMITERS(horizon, left, middle, right) then
13:                 INFERDELIMITER(left, right)
14:                 INFERMIDDLE(middle)                  ▷ Ignored if null
15:                 delimiters.REMOVE(left, middle, right)
16:                 break
17:         if LENGTH(delimiters) stayed the same then return Success
18:     return Success
```

The algorithm will start at line 4, by searching all new symbols and store them as delimiter candidates in `delimiters`. The function `sortedDelimiters` at line 11 will generate

all possible orders to put the delimiters into. It will also sort those combinations from most likely to unlikely. This is done by using the UCD `Bidi_Mirrored` property of paired delimiters (category `Z`) and checking if the order is coherent with the `Bidi_Class`.

Checking the result of the choice is very important. At line 12 the function `coherentDelimiters` checks if the delimiters allow for triple reduction and enforce restrictions.

Example 19. For example, a property cannot be wrapped in a container (except if part of parameters). 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 3.3 to $\mathbb{g}_0$. 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 (using the property `Name` starting with "left" and the property `Bidi_Class`). 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> :: = <parenthesis> | …
3 <parenthesis> ::= "(" [<naked_entity>] (<?MIDDLE> <naked_entity>)* ")"
4 <naked_entity> ::= <statement> | <entity>
```

Listing 3.3: Rules added to the current grammar for handling the new container for parenthesis

The rule at line 4 is added once and enables the use of meta-statements inside containers. It is the language pendant of the μ relation, allowing to wrap abstraction in a safe way.

Example 20. If we parse the expression `a = (b,c);`, we start by tokenizing it as `<ENTITY> <EQUAL> <SYMBOL><ENTITY><SYMBOL><ENTITY><SYMBOL> <EOS>` (ignoring whitespaces and comments). This means that the statement is 4 tokens too long to form a triple. This triggers a parsing error and then an evaluation using meta-rules. All the `<ID>` tokens are new symbols, but they don't have the same subtype. This means that candidate delimiters are (`(`), (`,`) and (`)`). To infer the correct combination, the left and right delimiters are found via their Unicode description. The comma is left to be inferred as the middle delimiter. The grammar is rewritten and the statement becomes `<ENTITY> <EQUAL> <container> <EOS>` which is a valid triple statement.

3.5.3.2 Parameters

If no viable container has been found, we proceed with the next meta-rule. This rule needs the first one to have been used at least once before being able to work. This meta-rule allows for parameterized entities using containers. A parameter is an ordered list of arguments, just like for functions. Algorithm 2 presents how we infer parameters from invalid statements.

Algorithm 2 Parameter meta-rule

```
1:  function PARAMETER(Entity[] statement)
2:      reduced = statement
3:      while LENGTH(reduced) >3 do
4:          for i from 0 to LENGTH(reduced) - 1 do
5:              if NAME(reduced[i]) not null and
6:  TYPE(reduced[i+1]) = Container and
7:  COHERENTPARAMETERS(reduced, i) then
8:                  param = INFERPARAMETER(reduced[i], reduced[i+1])
9:                  reduced.REMOVE(reduced[i], reduced[i+1])
10:                 reduced.INSERT(param, i)          ▷ Replace parameterized entity
11:                 break
12:         if LENGTH(statement) stayed the same  then return Success
13:     return Failure
```

This algorithm will search for extra containers in the statement. For each container, the function `coherentParameters` at line 7 will check if the container can be turned into a parameter for the preceding entity. In order to remove ambiguities, we disallow parameters on containers as well as using containers as properties.

Once a coherent parameter has been found, the container is removed from the statement and added as the preceding entity's parameter. To do so quicker the next time, the function `inferParameter` at line 8 adds that syntax as new rules as illustrated in listing 3.4, replacing `<?container>` with the name of the container used.

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

Listing 3.4: Rules added to the current grammar for handling parameters

Example 21. In this case we have to parse `f(x) = x;`. If we already have the parenthesis delimiter defined, it becomes `<ENTITY> <container> <EQUAL> <ENTITY> <EOS>`. Since the statement isn't a triple, we execute the meta-rules. The container rule finds no new symbols and fails. Then the parameter meta-rule will reduce the statement by bounding the first entity to the following container and mark it as a parameterized entity. This gives `<meta_entity> <EQUAL> <ENTITY> <EOS>`.

3.5.3.3 Modifiers

In some cases, using containers for parameters can become verbose. In order to make that task more concise, it is useful to add *syntactic sugar*. This term refers to a writing convenience that is actually equivalent to a longer or more complex notation. In our case, making containers optional come mainly from the use of modifiers. So this will comeS be our last meta-rule since it requires parameters to have been used at least once before. In algorithm 3 we explain the process of how the modifier notation is inferred.

In a very similar way as with algorithm 2, this algorithm will first determine if the proposed modifier is coherent. If the entity has been parameterized before, and the statement is valid after reduction, the syntax will be accepted. From the call

Algorithm 3 Modifier meta-rule

```
1:  function MODIFER(Entity[] statement)
2:      reduced = statement
3:      while LENGTH(reduced) >3 do
4:          for i from 0 to LENGTH(reduced) - 1 do
5:              if ν(reduced[i]) not null and
6:  ν(reduced[i+1]) not null and
7:  (ν(reduced[i]) is a new symbol or
8:  reduced[i] has been parameterized before) and
9:  COHERENTMODIFIER(reduced, i) then
10:                 mod = INFERMODIFIER(reduced[i], reduced[i+1])
11:                 reduced.REMOVE(reduced[i], reduced[i+1])
12:                 reduced.INSERT(mod, i)              ▷ Replace parameterized entity
13:                 break
14:         if LENGTH(statement) stayed the same  then return Success
15:     return Failure
```

of `inferModifier`, comes new rules explicated in listing 3.5. The call also adds the modifier entity to an inferred token `<MOD>`.

```
1 <meta_entity> ::= <?MOD> <ID>
2 <meta_property> ::= <?MOD> <ID>
```

Listing 3.5: Rules added to the current grammar for handling modifiers

Since it is most used for special entities like quantifiers, once used, the parent entity will take a polymorphic type. This means that type inference will not issue errors for any usage of them.

Example 22. With the input `!x = 0;` we parse `<SYMBOL> <ENTITY> <EQUAL> <LITERAL> <EOS>`. This cannot be a container since the new symbol is at the beginning without any mirroring possible. It cannot be a parameter since no container is present. But this will conclude with the modifier meta-rule as the new symbol precedes an entity. This becomes `<meta_entity> <EQUAL> <LITERAL> <EOS>` and becomes a valid statement.

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

3.5.4 Contextual Interpretation

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

When parsing, the rules `<entity>` and `<property>` will trigger the creation or retrieval of an entity. This mechanism will use the name of the entity to retrieve an entity with the same name in a given scope. If no such entity exists it is created and added to the current scope.

3.5.4.1 Naming and Scope

When parsing an entity, 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.

Example 23. In order to make this notion easier to understand, we start with its expected behavior. In figure 3.7 the process of variable qualification is illustrated. In order to become a variable, an entity must fulfill two conditions:

Figure 3.7: Example of scope resolution.

- Be used as a *parameter* in the statement.
- Be mentioned *twice* in the statement.

In our example, `f` has `x` as parameters and `x` is used in the statement inside the contained statement on the right hand of the first statement. Of course we suppose in this example that parentheses are delimiters and that (`;`) is the end of statement tokens.

An important nuance can be shown in the second statement. Indeed, `x` is also a variable but since it is a different global level statement, *it is not resolved as the same variable as the previous* `x`. This means that both variables are **independent**.

In the third statement, `x` is only mentioned once and therefore is now a global symbol. It is still independent from the two previous statements.

A scope is the reach of an entity's direct influence. It affects the naming relation by removing variable names. 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 also makes the use of variables possible, akin to RDF blank nodes.

The scope of an entity has three special values :

- *Variable*: This scope restricts the scope of the entity to only the other entities in its scope.
- *Local*: This scope is temporarily bound to a given entity during the parsing. This scope is limited to the statement being interpreted.
- *Global*: This scope means that the name has no scope limitation.

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

détailler l'algo

Algorithm 4 Determination of the scope of an entity

```
1:  function INFERSCOPE(Entity e)
2:      Entity[] reach = []
3:      if : (e) = S then
4:          for all i ∈ χ(e) do reach.APPEND(INFERVARIABLE(i))      ▷ Adding scopes nested in statement e
5:      for all i ∈ μ*(e) do reach.APPEND(INFERVARIABLE(i))         ▷ Adding scopes nested in container e
6:      if ∃ρ(e) then
7:          Entity[] param = INFERSCOPE(ρ(e))
8:          for all i ∈ param do param.REMOVE(INFERSCOPE(i))        ▷ Remove duplicate scopes from
    parameters
9:          for all i ∈ param do reach.APPEND(INFERVARIABLE(i))     ▷ Adding scopes from paramters of e
10:     SCOPE(e) ← reach
11:     if GLOBAL ∉ SCOPE(e) then SCOPE(e) ← SCOPE(e) ∪{LOCAL}
    return reach
12: function INFERVARIABLE(Entity e)
13:     Entity[] reach = []
14:     if LOCAL ∈ SCOPE(e) then
15:         for all i ∈ SCOPE(e) do
16:             if ∃e_p ∈ U : ρ(p) = i then                        ▷ e is already a parameter of another entity e_p
17:                 SCOPE(e) ← SCOPE(e) \{LOCAL}
18:                 SCOPE(e_p) ← SCOPE(e_p) ∪ SCOPE(e)
19:                 SCOPE(e) ← SCOPE(e) ∪{VARIABLE, p}
20:         reach.APPEND(e)
21:         reach.APPEND(SCOPE(e))
    return reach
```

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 since it is referenced twice in the same statement.

3.5.4.2 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 entities 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.

3.5.5 Structure as a Definition

The derivation feature on its own does not allow to define most of the native properties. For that, one needs a light inference mechanism. This mechanism is part of the

default inference engine. An *inference engine* is the term that describes an algorithm used in ontologies to infer new statements from an explicit set of statements known as an ontological database.

In our case, this engine only works on the principle of structure as a definition. Since all names must be neutral from any language, that engine cannot rely on classical mechanisms like configuration files with keys and values or predefined keywords.

To use SELF correctly, one must be familiar with the native properties and their structure or implement their own inference engine to override the default one.

3.5.5.1 Quantifiers

In SELF quantifiers differ from their mathematical counterparts. The quantifiers are special entities that are meant to be of a generic type that matches any entities including quantifiers. There are infinitely many quantifiers in SELF but they are all derived from a special one called the *solution quantifier*. We mentioned it briefly during the definition of the grammar $\mathbb{g}_0$. It is the language equivalent of $\mu^\bullet$ and is used to extract and evaluate reified knowledge (see section 3.5.1.2).

Example 24. The statement `bob is <SOLVE>(x)` will give either a default container filled with every value that the variable `x` can take or if the value is unique, it will take that value. If there is no value it will default to `<NULL>`, the exclusion quantifier.

How are other quantifiers defined? We use a definition akin to Lindstöm quantifiers (1966) which is a generalization of counting quantifiers (Gradel *et al.* 1997). Meaning that a quantifier is defined as a constrained range over the quantified variable. We suppose five quantifiers as existing in SELF as native entities.

- The **solution quantifier** `<SOLVE>` noted $\S$ in classical mathematics, turns the expression into the possible value range of its variable. It is like replacing it by the natural expression "those x that".
- The **universal quantifier** `<ALL>` behaves like $\forall$ and forces the expression to take every possible value of its variable.
- The **existential quantifier** `<SOME>` behaves like $\exists$ and forces the expression to match *at least one* arbitrary value for its variable.
- The **uniqueness quantifier** `<ONE>` behaves like $!\exists$ and forces the expression to match *exactly one* arbitrary value for its variable.
- The **exclusion quantifier** `<NULL>` behaves like $\neg\exists$ and forces the expression not to match the value of its variable.

The last four quantifiers are inspired from Aristotle's square of opposition (D'Alfonso 2011) as illustrated in figure 3.8.

In SELF, quantifiers are not always followed by a quantified variable and can be used as a value. In that case the variable is simply anonymous. We use the exclusion quantifier as a value to indicate that there is no value, sort of like `null` or `nil` in programming languages.

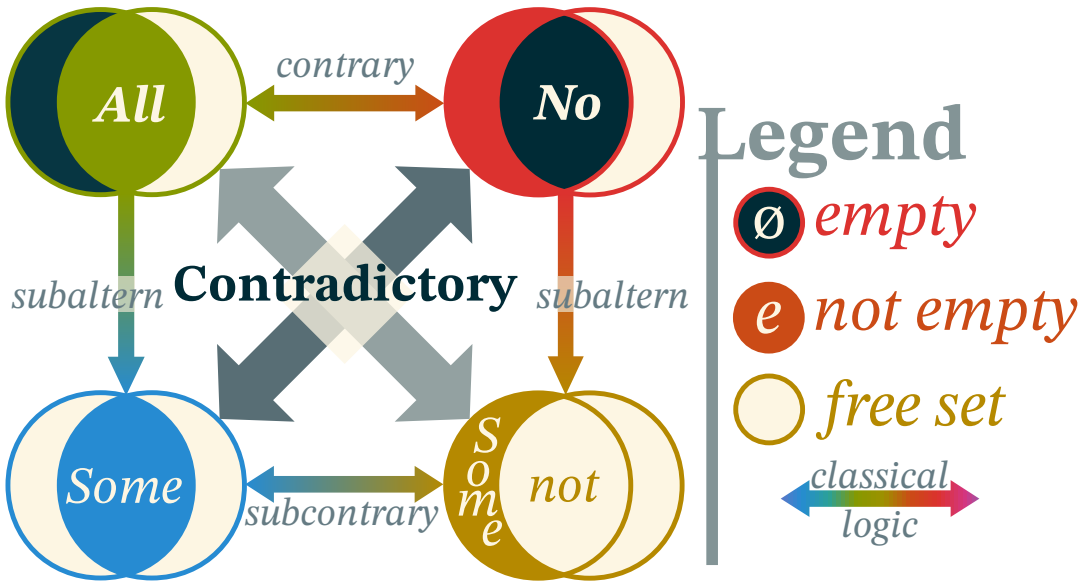


Figure 3.8: Aristotle's square of opposition

Example 25. If we want to express the fact that a glass of water is not empty we can write either `glass contains ~(~);` or `glass ~(contains) ~` with `<NULL> = ~`. This shows that `<NULL>` is used for negation and to indicate the absence of value.

This property is quite handy as it requires only one symbol and allows for complex constructs that are difficult to explain using available paradigms.

In listing 3.6, we present an example file that is meant to define most of the useful native properties along with default quantifiers.

```
* =? ;
?(x) = x; //Optional definition
?~ = { };
?_ ~(=) ~;
?!_ = {_};

(*e, !T) : (e :: T); *T : (T :: Type);
*T : (Entity / T);

:: :: Property(Entity, Type);
(___) :: Statement;
(~, !, _, *) :: Quantifier;
( )::Group;
{ }::Set;
[ ]::List;
< >::Tuple;
Collection/(Set,List,Tuple);
0 :: Integer; 0.0::Float;
'\0'::Character; ""::String;
Literal/(Boolean, Integer, Float, Character, String);

(*e, !(s::String)) : (e named s);
(*e(p), !p) : (e param p);
```

```
24 *(s p o):(((s p o) subject s),((s p o) property p),((s p o) object o));
```

Listing 3.6: The default lang.w file.

At line 1, we give the first statement that defines the solution quantifier's symbol. The reason this first statement is shaped like this is that global statements are always evaluated to be a true statement. Since domains are sets of statements, this means that anything equaling the solution quantifier at this level will be evaluated as a domain. This is because the entity is a domain **by structure**. If it is a single entity then it becomes synonymous to the entire SELF domain and therefore contains everything. We can infer that it becomes the universal quantifier.

If it is a string literal, then it must be either a file path or URL or a valid SELF expression.

Example 26. Using the first statement, we can include external domains akin to the `import` directive in Java. Writing `"path/lang.w" = ? ;` as a first statement will make the process parse the file located at `path/lang.w` and insert it at this spot.

All statements up to line 5 are quantifiers definitions. On the left side we got the quantifier symbol used as a parameter to the solution quantifier using the operator notation. On the right we got the domain of the quantifier. The exclusive quantifier has as a range the empty set. For the existential quantifier we have only a restriction of it not having an empty range. At last, the uniqueness quantifier got a set with only one element matching its variable (noting that anonymous variables do not match other anonymous variables necessarily in the same statement).

In listing 3.6 the type hierarchy can be illustrated by the figure 3.9. It shows how the type `Entity` is the parent of all types. This figure is separated by two axes of symmetry. The vertical separation concern abstraction. Types on the left are used for the Meta relation. The horizontal line distinguishes valuation. Terms on top are externally valued (valued by the context) and terms on the bottom are intrinsically valued (valued by their definition).

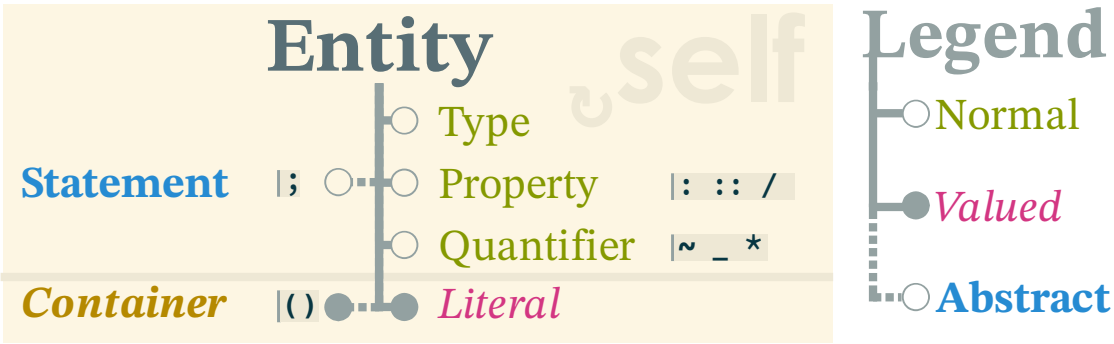


Figure 3.9: Hierarchy of types in SELF

3.5.5.2 Inferring Native Properties

All native properties can be inferred by structure using quantified statements. Here is the structural definition for each of them:

- $=$ (at line 1) is the equality relation given in the first statement.
- $\subseteq$ (at line 8) is the first property to relate a particular type of all types. That type becomes the entity type.
- $\mu^{\bullet}$ (at line 1) is the solution quantifier discussed above given in the first statement.
- μ is represented using containers.
- ν (at line 22) is the first property affecting a string literal uniquely to each entity.
- ρ (at line 23) is the first property to effect to all entities a possible parameter list. Affect
- $:$ (at line 7) is the first property that matches every entity to a type.
- χ (at line 24) is the first property to match for all statements.

We limit the inference to one symbol to eliminate ambiguities and prevent accidental redefinition of native properties. This also improves performance as the inference is stopped after finding a first matching entity that can be used programmatically using a single constant.

3.5.6 Extended Inference Mechanisms

In this section we present the default inference engine. It is quite limited since it is meant to be universal and the goal of SELF is to provide a framework that can be used by specialists to define and code exactly what tools they need.

Inference engines need to create new knowledge but this knowledge shouldn't be simply merged with the explicit user provided domain. Since this knowledge is inferred, it is not exactly part of the domain but must remain consistent with it. This knowledge is stored in a special scope dedicated to each inference engine. This way, inference engines can use defeasible logic or have dynamic inference from any knowledge insertion in real time.

3.5.6.1 Type Inference

Type inference works on matching types in statements. The main mechanism consists in inferring the type of properties in a restrictive way. Properties have a parameterized type with the type of their subject and object. The goal is to make that type match the input subject and object.

For that we start by trying to match the types. If the types differ, the process tries to reduce the more general type against the lesser one (subsumption-wise). If they are incompatible, the inference uses some light defeasible logic to undo previous inferences. In that case the types are changed to the last common type in the subsumption tree.

However, this may not always be possible. Indeed, types can be explicitly specified as a safeguard against mistakes. If that's the case, an error is raised and the parsing or knowledge insertion is interrupted.

3.5.6.2 Instanciation

instanTiation

Another inference mechanism is instantiation. Since entities can be parameterized, they can also be defined against their parameters. When those parameters are variables, they allow entities to be instantiated later.

Since entities are immutable, updating their instance can be quite tricky. Indeed, parsing happens from left to right and therefore an entity is often created before all the instantiation information are available. Even harder are completion of definition in several separate statements. In all cases, a new entity is created and then the inference realize that it is either matching a previous definition and will need to be merged with the older entity or it is a new instance and needs all properties duplicated and instantiated.

realizeS

TO BE

This gives us two mechanisms to take into account: merging and instanciating.

Merging is pretty straightforward: the new entity is replaced with the old one in all of the knowledge graph. Containers, parameterized entities, quantifiers and statements must be duplicated with the correct value and the original destroyed. This is a heavy and complicated process but seemingly the only way to implement such a case with immutable entities.

Instanciating is similar to merging but even more complicated. It starts with computing a relation that maps each variable that needs replacing with their grounded value. Then it duplicates all knowledge about the parent entity while applying the replacement map.

3.6 Example

In the following section, a use case of the framework will be presented. First we have to explain a few notions.

3.6.1 Modality of Statements

In the field of logic there exists one special flavor of it called *modal logic*. It lays the emphasis upon the qualifications of statements, and especially the way they are interpreted. This is a very appropriate example for SELF. The modality of a statement acts like a modifier, it specifies a property regarding its plausibility, origin or validity.

Example 27. In the figure 3.10, we present a case of three persons gossiping, Alice, Becky and Carol. The presentation is inspired by the work of Schwarzentruber (2018). Here is a list of the statements in this example:

- Alice said to Becky that Carol should *probably* change her style from C_1 to C_2.
- Becky said to Alice that she finds the Carol's style *usually* good.
- Alice told Carol that Becky told her that she should *sometimes* change her style to C_2.

The following statement can be inferred:

- Carol *possibly* thinks that Becky thinks that the style C_2 is *often* good.

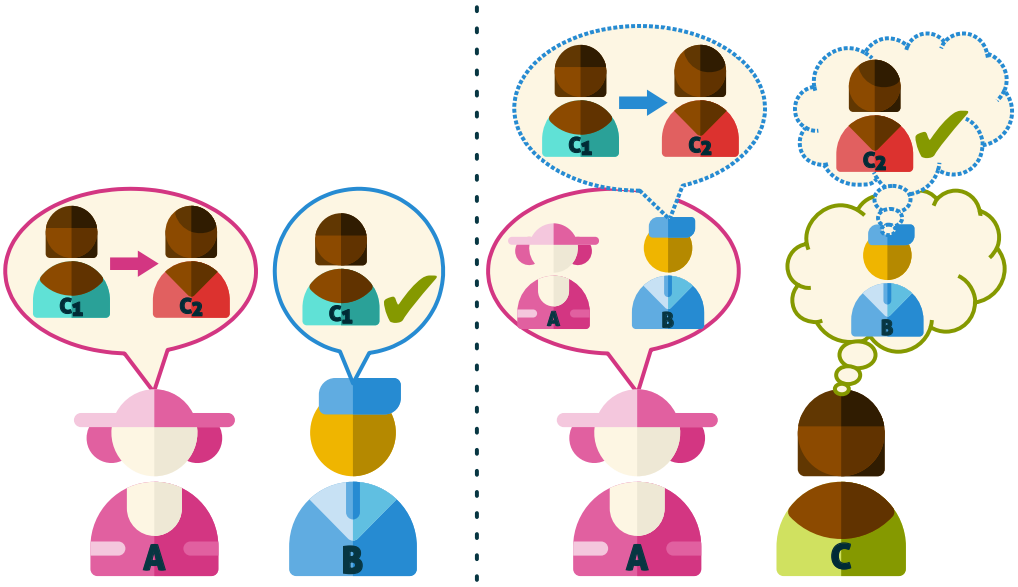


Figure 3.10: Example of modal logic propositions: Alice gossips about what Beatrice said about Claire

In the example, all modalities are *emphasized*. One can notice an interesting property of these statements in that they are about other statements. This kind of description is called *higher order knowledge*.

3.6.2 Higher order knowledge

SELF is based on the ability to easily process higher order knowledge. In that case the term *order* refers to the level of abstraction of a statement (Schwarzentruber 2018). For such usages, a hypergraph structure is a clear advantage in terms of expressivity and ease of manipulation of those statements. This is due to the higher dimensionality of sheaves (and by extension hypergraphs) that makes meta-statement as simple to express as any other statement. This chain of abstractions using meta-statements is where the higher order knowledge is encoded.

Example 28. We present the previous example using RDF (in listing 3.7) and SELF (in listing 3.8) to describe knowledge of the gossip.

```
@prefix : <http://genn.io/self/gossip#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@base <http://genn.io/self/gossip> .

<http://genn.io/self/gossip> rdf:type owl:Ontology ;
                             owl:imports rdf: .

:modality rdf:type owl:AnnotationProperty ;
          rdfs:range :Modality .

:told_a rdf:type owl:ObjectProperty .
:told_b rdf:type owl:ObjectProperty .
:told_c rdf:type owl:ObjectProperty .
:Modality rdf:type owl:Class .
:good rdf:type owl:NamedIndividual .
:is rdf:type owl:NamedIndividual ,
             rdf:Property .
:probably rdf:type owl:NamedIndividual ,
                   :Modality .
:s1 rdf:type owl:NamedIndividual ,
             rdf:Statement ;
    rdf:object :c2 ;
    rdf:predicate :worsethan ;
    rdf:subject :c ;
    :modality :probably .
:s2 rdf:type owl:NamedIndividual ;
    rdf:object :good ;
    rdf:predicate :is ;
    rdf:subject :c ;
    :modality :usually .
:s3 rdf:type owl:NamedIndividual ,
             rdf:Statement ;
    rdf:object :s4 ;
    rdf:predicate :told_a ;
    rdf:subject :b .
:s4 rdf:type owl:NamedIndividual ,
             rdf:Statement ;
    rdf:object :c2 ;
    rdf:predicate :should ;
    rdf:subject :c ;
    :modality :sometimes .
:should rdf:type owl:NamedIndividual ,
                 rdf:Property .
:sometimes rdf:type owl:NamedIndividual ,
                    :Modality .
:told_a rdf:type owl:NamedIndividual ,
                 rdf:Property .
:usually rdf:type owl:NamedIndividual ,
                  :Modality .
:worsethan rdf:type owl:NamedIndividual ,
                    rdf:Property .
:a rdf:type owl:NamedIndividual ;
   :told_b :s1 ;
   :told_c :s3 .
:b rdf:type owl:NamedIndividual ;
```

```
60    :told_a :s2 .
61 :c rdf:type owl:NamedIndividual .
62 :c2 rdf:type owl:NamedIndividual .
```

Listing 3.7: RDF representation of the gossip example

```
1 "lang.s" = ? ;
2 a told(b) probably(c worsethan ctwo);
3 b told(a) usually(c is good);
4 a told(c) (b told(a) sometimes(c should ctwo));
```

Listing 3.8: SELF representation of the gossip example

It is obvious that the SELF version is an order of magnitude more concise than RDF to express modal logic. The 4 lines of SELF are **equivalent** to the 62 lines of RDF. In the RDF version we use the reified statements `:s1`, `:s2`, `:s3` and `:s4` along with a `:modality` annotation to express high order knowledge and modalities. In SELF, everything is inferred by structure and one can start exploiting their database right away.

3.7 Conclusion

In this chapter, we presented a new endomorphic metalanguage for knowledge description. This language along with its framework allows for extended expressivity and higher order knowledge. This framework was needed to overcome the limitations of classical knowledge representation tools, mainly in order to encode hierarchical planning domains into human and computer-readable text.

In the following chapter we will show an application of this framework to encode planning domains. This application allows to transpose a general planning framework into a specialized language using SELF.