Towards a Mathematical and Computational Theory of Meaning in Natural Languages

Benoît Sauzay, Gaëll Guibert, Jean-Pierre Desclés
Université Paris-Sorbonne, STIH-LaLIC
28, rue serpente, 75006, Paris, France
benoit.sauzay@gmail.com, gaell.guibert@wanadoo.fr, Jean-pierre.Descles@paris4.sorbonne.fr

Abstract

In one hand, the meaning of natural languages is often described with basic semantic features and a boolean composition of these features. However, this approach is not sufficient to describe more deeply the meaning of linguistic units. In the other hand, the semantic of computer languages often starts from Church's λ-calculus and walks up to more abstract levels. In this communication, is introduced a new general computational approach of the representation of meanings for high level languages (natural languages and programming languages), in working from the paradigm of compilation in computer sciences. In this compilation paradigm, the expressions of a high level symbolic language are changed in representations by means of intermediary levels, to hit formal representations directly compatible with the material structures of a machine or of a brain. From this analogy, expressions of a natural language can be analyzed by different metalinguistic levels of representations linked to each other by changing representation processes. The communication will give examples of representation changing between expressions of English and representations of meanings expressed by algebra of "*treilles*".

Introduction

In the field of artificial intelligence, questions arise: Can computers manage semantic representations as human do? Does the understanding of such representations rely on some operational framework? To some extent, how can we consider the construction of new representations by the use of existing ones? This is definitely a key property of intelligent systems. These questions will lead our reflection and argumentation about relationships between high levels representations and technical systems with compilation. By compilation we mean representation or structural changes between levels.

Issues and Perspectives in Linguistics and Computation

In a more technical point of view: can we make an assumption on the building and evaluation of semantic representations without restricting ourselves to synonymy relations? Can we propose a mathematical and computational framework that would be able to construct the meaning of significant units like those handled by natural languages? Such that, they could operate as programming languages do for "functions". Then, this framework would be an assumption to explain computations performed by the brain and a research direction to make them work effectively on a machine.

A beginning of the answer is: thinking in terms of compilation rather than making a direct correspondence between a particular area in the brain and a word or meaning. Compilation involves a set of properties that would be subject to a set of transformations between different levels. In that purpose, we start, in mathematics, with a formal description of our general computation system. The compilation operates inside the mathematical framework itself: the first change of representation is carried out between a particular category, $\underline{\underline{T}}[\Sigma]$ (Desclés,1980), using commutative diagrams, and a linear algebraic structure. The second one is performed between this previous algebraic structure and a non-linear algebraic structure, called "*treille*". The latter is a representation in the plan and is sufficient to express by itself operational semantic representations. In natural languages, we make the assumption that a similar process is involved by the cognition: from high level semantico-cognitive representations, or schemes, to utterances and vice versa.

Some existing approaches use to adopt a bottom-up perspective. They start from observable data and elaborate some levels of abstraction in the hypothesis that this abstract or formal level is a suitable representation of the data. On the one hand, semantic with features presents this ascending method. From properties of any extension, it lists the characteristics until building minimal abstract unities of semantic description called features, representing the object at an abstraction level. B. Pottier edified analysis with *semes* for semantic comprehension in linguistics (1963; 1985, p. 63). Some minimal semantic features can be distinguished and then compared, by similarities and differences. These semantic principles remain today, for semantics and synonymy relations. On the other hand, in computer software, high levels representations are compiled with a similar process down to lower levels until being carried out by the material of the machine: most of the functional programming languages rely on the λ-calcul and develop complex type systems in order to get consistency. The F system from J.-Y. Girard, with dependent types, is definitely the most elaborate of them.

After having discussed two points of the state of the art in a first part, we present our formal system with its different levels of representation in a second part, and in a third part we will give examples of semantic representations based on our mathematical and computational framework in the compilation paradigm.

Formal Systems Compilation: from the $\underline{\underline{T}}[\Sigma]$ category to the algebra of "*treilles*"

Our purpose is to define here, a general consistent computational framework without any external references or dependencies. We start from the Category Theory, and thanks to "abstract compilation" we change the representations of our formal system down to an elementary writing rule and an evaluation rule ("reading over nodes of the product"), to build up our categorical computational system (Desclés 1980). This process of changing levels of representation is exactly the same as those performed by compilers in programming.

Categorical Computational System: $\underline{\underline{T}}[\Sigma]$

Our theoretical framework is the Category Theory. Categories are "objects of thoughts" that target the expression of properties between objects rather than the objects themselves. They are mainly represented with arrows and diagrams; these arrows are identity and composition. In the field of computation, the general case may be defined by the passage from a tuple of heterogeneous data to another, whatever the data (e.g. numbers, vectors, strings, functions…). In the state of the art, Σ-algebra in cartesian closed categories or the typed λ-calculus from A. Church with the cartesian product achieve the general case. However another approach consists in working only with sorts or types of data rather than data.

Thus we split the notion of computation into, on the one hand, operators σ, and, on the other hand, operations σ•. We will mainly focus here on the operator side. We point out that, an association between operators and operations thanks to a particular arrow called a functor $\underline{A}$, build up again the notion of function. Considering that μ, ν are monoidal categories and X^ν. X^μ are categories of sets, our system is sum up by the diagram:

$$\begin{tikzcd}
\mu \arrow[r, "\underline{A}(\mu)"] \arrow[d, "\sigma"', "(\text{operator})"] & X^\mu \\
\nu \arrow[r, "\underline{A}(\nu)"'] & X^\nu \arrow[u, "\sigma\bullet"', "(\text{operation})"]
\end{tikzcd}$$

$$\underline{A}(\sigma) = \sigma^\bullet \in \mathrm{Hom}(X^\nu, X^\mu)$$

Fig. 1: The $\underline{\underline{T}}[\Sigma]$ category

Operators are defined in fig. 1 as a specific arrow. They are applied on sorts structured in monoid (*e.g.* μ, ν with $\mu = s_1 \ldots s_n$ and $\nu = s'_1 \ldots s'_n$) and operations on sets of typed data, respectively X^μ and X^ν. X^μ means a set of objects of type $\mu = s_1 \ldots s_n$ where s_i is a sort. The arrow of operators is in the opposite direction than the arrow of operations. This property is called contravariance, and is carried out by our contravariant functor $\underline{A}$. This elementary property is the core of our general computational system.

There are two transformations on operators, the T-operations: the "greffe" ("grafting" or a functional composition of operators) is comparable to composition on sets. The "intrication" (a tensorial product of operators) is specific to our computational approach and does not exist in other systems. Specific operators, without associated operations, called "coprojectifs", are also introduced for co-projective operations (permutations, deletion … of arguments, and introduction of fictive arguments).

Formal definitions of the T-operations are now addressed.

Greffe in $\underline{\underline{T}}[\Sigma]$: o

This transformation, in the left of the fig. 2, is associative. Each arrow is an operator that has an associated operation (arrow on sets in the opposite direction) on the right of the figure.

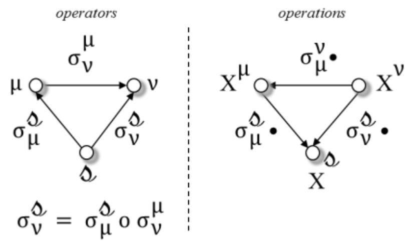


Fig. 2: Greffe in $\underline{\underline{T}}[\Sigma]$

Intrication in $\underline{\underline{T}}[\Sigma]$: ⊗
Intrication defines a “formal” parallelism: the tensorial product of *n* operators (or multi-operators) can be “reduced” to one arrow or multi-operator. Multi-operators are intrications of simple operators.

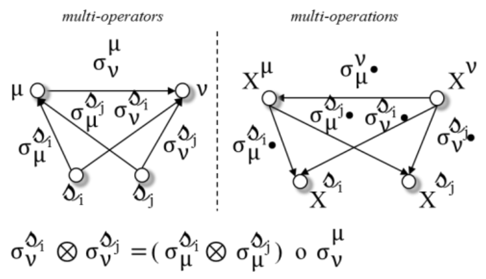


Fig. 3: Intrication in $\underline{\underline{T}}[\Sigma]$

The intrication can be realized by the *greffe* of particular *coprojectifs* if we consider that all structures are “intricated”. These categorical representations are not directly usable in programming, particularly intrication. Composition in programming languages is more often defined on the operation side, using variables on sets.

Back to our compilation paradigm, we need now to change the representation for our system. Moving towards algebraic structures is a possible solution to make it operational on structures and then machines down to more concrete representations.

Linear Algebraic Computational System: T[Σ]-algebra
The T[Σ]-algebra is the set of arrows (that this called an “algebraic theory” by Lawvere and Birkoff) of our $\underline{\underline{T}}[\Sigma]$ category. It generalizes directly the Σ-algebra.

An algebraic structure mainly relies on product and sum “operations”. A key point of this approach is that product and sum won’t be the primitives to build up *greffe* and *intrication*, but on the contrary, *greffe* and *intrication* build up product and sum “operations” (If the latter can be compared with intersection and union, there is no direct connection with lattices). We will develop this further with a nonlinear algebraic structure: the *treilles*. This is why, previous categorical definitions can be re-written within the current algebraic structure. This new writing is a change of representation under the category level.

Operator or multi-operator
An operator or multi-operator σ, of name ν, of input type $\nu = s'_1 \ldots s'_n$ and of output type $\mu = s_1 \ldots s_n$ is given as follow:

$$\boldsymbol{\sigma}_{\boldsymbol{\nu}}^{\boldsymbol{\mu}} \equiv <\boldsymbol{\nu}, <\boldsymbol{\mu}, \boldsymbol{\nu}>> \in \mathbf{T}[\Sigma]_\nu^\mu$$

where $T[\Sigma]_\nu^\mu$ is a set of arrows which are multi-operators. The operator has in input, a word ν of sorts, and in output, a word μ of sorts.

Greffe in the T[Σ]-algebra : o
It is defined by the following rule:
To each pair:

$$< \boldsymbol{\sigma}_\rho^{\boldsymbol{\mu}}, \boldsymbol{\sigma}_{\boldsymbol{\nu}}^\rho > \in T[\Sigma]_\rho^\mu \text{ x } T[\Sigma]_\nu^\rho$$

we associate the unique corresponding element:

$$\boldsymbol{\sigma}_{\boldsymbol{\nu}}^{\boldsymbol{\mu}} = \boldsymbol{\sigma}_\rho^{\boldsymbol{\mu}} \circ \boldsymbol{\sigma}_{\boldsymbol{\nu}}^\rho \in T[\Sigma]_\nu^\mu$$

This very simple rule forms the core of our system. The consistency of it is immediate: operators are theorems in our framework, as sorts or types are directly handled and not variables.

Intrication in the T[Σ]-algebra : ⊗
To each tuple:

$$< \boldsymbol{\sigma}_{\boldsymbol{\nu}}^{\boldsymbol{s_1}}, \ldots, \boldsymbol{\sigma}_{\boldsymbol{\nu}}^{\boldsymbol{s_n}} > \in \mathbf{T}[\Sigma]_{\boldsymbol{\nu}}^{\boldsymbol{s_1}} \textbf{ x} \ldots \textbf{x } \mathbf{T}[\Sigma]_{\boldsymbol{\nu}}^{\boldsymbol{s_n}}$$

we associate the unique corresponding element:

$$\boldsymbol{\sigma}_{\boldsymbol{\nu}}^{\boldsymbol{\mu}=\boldsymbol{s_1}\ldots\boldsymbol{s_n}} = \boldsymbol{\sigma}_{\boldsymbol{\nu}}^{\boldsymbol{s_1}} \otimes \cdots \otimes \boldsymbol{\sigma}_{\boldsymbol{\nu}}^{\boldsymbol{s_n}} \in \mathbf{T}[\Sigma]_\nu^\mu$$

In order to write linear expression of operators and operations we need a rule to express the application of operators on operands to build up results.

Operator and operand
This is achieved by the following rule:

$$\frac{\dfrac{\sigma_\nu^\mu = \sigma_\nu^{s_1} \otimes \cdots \otimes \sigma_\nu^{s_m} \quad < \boldsymbol{x}_{s'_1}, \ldots, \boldsymbol{x}_{s'_m} > \in X^{\nu = s'_1 \ldots s'_m}}{\sigma_\mu^\rho = \sigma_\mu^{s''_1} \otimes \cdots \otimes \sigma_\mu^{s''_p} \quad < \boldsymbol{y}_{s_1}, \ldots, \boldsymbol{y}_{s_m} > \in X^{\mu = s_1 \ldots s_m}}}{\ldots \quad < \boldsymbol{z}_{s''_1}, \ldots, \boldsymbol{z}_{s''_p} > \in X^{\rho = s''_1 \ldots s''_p}}$$

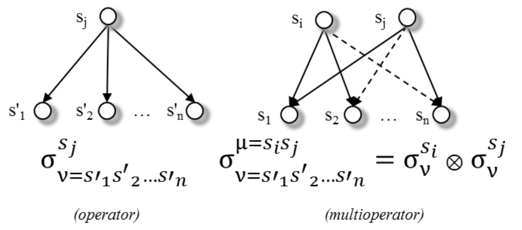
Where each *y* or *z* resulting values, are the result of the action of the operation $\boldsymbol{\sigma}_{\boldsymbol{\nu}}^{\boldsymbol{s_i}} \bullet$ associated to the corresponding $\boldsymbol{\sigma}_{\boldsymbol{\nu}}^{\boldsymbol{s_i}}$ operator, on $< x_1, \ldots x_m >$ tuple of values.

Nonlinear Algebraic Computational System: *Treilles*
The present representation is nonlinear as it is defined in the plan or $\mathbb{R}^2$. It comes from the compilation process from the previous algebra; it is called algebra of “*treilles*”, and is isomorphic to the T[Σ]-algebra. A *treille* diagram generalizes the algebraic structure of two-oriented trees (used in computer science and in linguistic); a *treille* is an algebraic version of DOAG often used in programming; it allows representing connections between two occurrences of a same unit without using linked variables, pointers or even stacks.

The meaning of an expression (for instance a sentence) of a high level language is represented by a formal expression generated from a schema described in the form of a *treille*. The changing representation process from high level to an associated *treille* is explicitly defined by specific formal operations of compilation.

We proceed here as we did for the $\underline{\underline{T}}[\Sigma]$ category and the T[Σ]-algebra. We start with the definition of an operator and continue with the definitions of intrication and *greffe*.

Operator and multi-operator in *treilles*



s_j
s'_1 s'_2 … s'_n
$\sigma^{s_j}_{\nu=s'_1 s'_2 \ldots s'_n}$
(operator)
s_i s_j
s_1 s_2 … s_n
$\sigma^{\mu=s_i s_j}_{\nu=s'_1 s'_2 \ldots s'_n} = \sigma^{s_i}_{\nu} \otimes \sigma^{s_j}_{\nu}$
(multioperator)

Fig. 4: Operator and Multi-operator

Treilles have two "orientations": a hierarchical order (vertical) and a succession order (horizontal) that organizes all immediate successors of a specific node. We emphasis that dead links (dotted arrows) are explicit in *treilles* and implicit in trees, and that the annotations of nodes are sorts $s_1 \ldots s_n$ and not variables *x*, *y*, *z* with variation domains.

Intrication in the *treilles*: ⊗

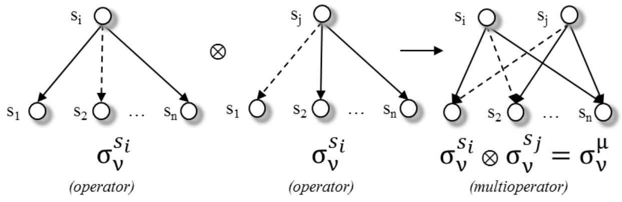
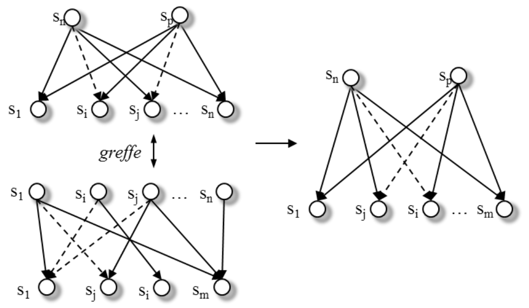


Fig. 5: Intrication in the treilles

Intrication ⊗ allows building a multi-operator thanks to two operators or multi-operators. In the figure above, we have on the left two operators and on the right one resulting multi-operator. The operation succeeds if and only if the two operators have the same sorts on the terminal nodes (s_1, $s_2 \ldots s_n$), otherwise it is not defined.

Greffe in the *treilles* : o

The *greffe* succeeds if and only if the terminals (sorts) of the upper operator are the same as the roots of the lower operator. Like the intrication, the *greffe* is also associative.



s_n s_p
s_1 s_i s_j … s_n
greffe
s_1 s_i s_j … s_n
s_1 s_j s_i s_m
s_n s_p
s_1 s_j s_i … s_m

Fig. 6: Greffe in the treilles

The *greffe* is not defined everywhere: we can *greffe* two multi-operators if they have the same number of respectively leaves and roots. The intermediate nodes (fig. 6) are deleted by the *greffe*. This can be "compared" to other representations, such as a simultaneous substitution of linked variables in the λ-calculus or the combinator **B** of the combinatory logic from H. Curry.

Reading over the nodes of the product

We did all these transformations in order to get a simple "evaluation rule" for the system. The result comes from the reading on the *greffes* of a *treille*:

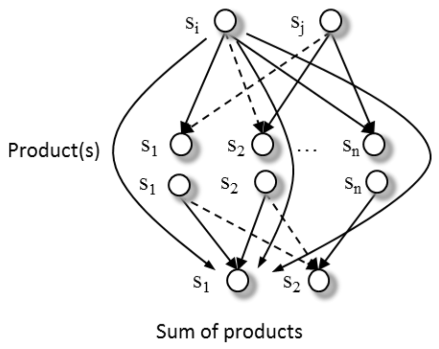


Fig. 7: Reading over the nodes of the product

The product is then clarified: the intermediary nodes are forgotten inside the structure when the calculus passes over them. This "reading" connects inputs and outputs of the operator without reduction of the structure.

Connections with cognitive representations are close to this approach. Significant units of natural languages are markers of such structured relations between "objects". Examples, using different representations but isomorphic properties, will be given now, where verbs are operators and semantic schemes at meaning level. The interpretation of previous structures in a particular domain is then discussed.

Natural languages applications

Linguistic utterances own semantic values. The top-bottom method described previously is reiterated here in the semantic field. The semantic of expressions from natural languages or programming languages is defined as the interpretation of one syntax by another syntax (Sauzay, 2013). In programming, interpretation means to compute or give the means for a calculus. Interpreting a sentence consist to express a calculus with all the "values": starting and ending situations at meaning level with relations between both. In other words, interpreting a sentence consists in building the corresponding *treille*.

The correspondence between syntactic or sentence and semantic levels takes only place by the mean of rules from the mathematical framework: rules of inversion, deletion, duplication. All linguistic forms are composition of operators and operands from the different levels (*e.g.* syntactic, grammatical, semantic and cognitive), handling sorts and composing them to produce new semantic values.

In this section, we present semantico-cognitive schemes, and corresponding *treilles*. We start with simple examples and finish with more complex ones.

To Go Outside

For example, if we have the sentence:

John goes outside the house.

The question is: can we represent the semantic of the predicate *to go outside* by a calculus? Can a structure, interpreted in a specific domain, express the semantic of the predicate *to go outside* (considering only the predicate, without aspectual values)? If predicates are operators on sorts, the answer is immediate: the predicate is represented by an intrication of *treilles*.

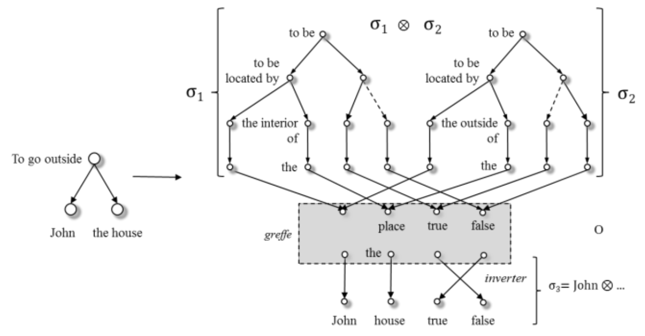


Fig. 8: Semantic representation of "To go outside"

We just need to find linguistic operators to "fill" the values of the *treilles*, like a lexicon describes words with other words, or like in programming, "functions" describe other "functions" in a recursive way.

The transformation can be done by a set of compositions (the details are given in Sauzay 2013).

The first operator σ_1 can be paraphrased by *John is located by the interior of the place: the house, is true*. The second, σ_2, can be paraphrased by *John is located by the outside of the place: the house, is false*. A composition with an inverter changes simultaneously, or in parallel, the semantic values of the two *treilles*. What was true becomes false and *vice versa* without any synonymy relation.

In programming, in a "first class" point of view, we have two "values" of *treilles* or operators which are applied to boolean sorts (and not boolean values as we use to).
The next example introduces some control with specific operators.

To Kill

However, if we take the example:

The hunter killed the deer

We deduce immediately that:

The deer is dead.

The question is: how can we build up a calculus that manages the representations associated to the predicates "to kill" and "to die"? These representations are neither synonyms nor independents. As we did for the predicate to go outside, we present the associated *treille* and transformation for the predicate to kill.

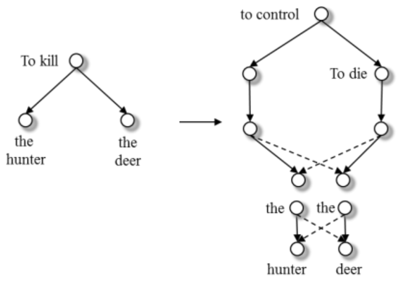


Fig. 9: Transformation associated with "To Kill"

We add some compositions inside the *treille* to allow a calculus on "to die". We give the expanded view:

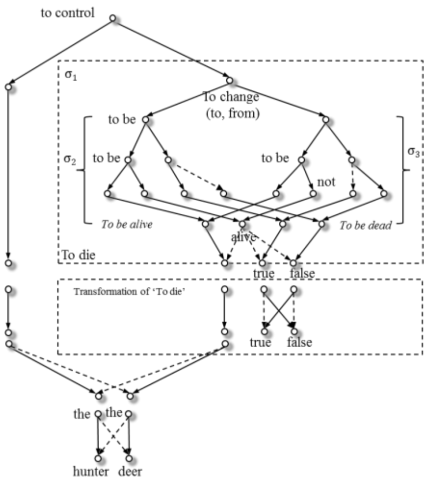


Fig. 10: Development of the predicate "To die"

In "To die", there is a sequence of compositions, built up thanks to the predicate "to be alive" and "to be dead" (not "to be alive"), and composed with "to change".

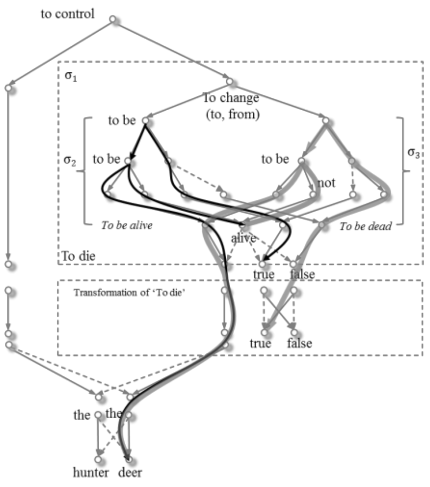


Fig. 11: Calculus over "To die"

The whole is composed with "to control" in order to build the predicate "to kill". It is clear that we don't have semantic features anymore.

As we described the reading of the nodes of the product in the mathematical framework, fig. 11 can be read as the interpretation or result on the *treille*. This calculus could be developed adding aspectual values. Thus, the calculus would be able to answer questions like: *Was the deer alive before being killed by the hunter*? Or, *Will the deer be dead after being killed by the hunter*?

Further examples: to enlarge, to spread, to increase

Complex predicates can be represented by multi-operators. For example, the predicate "to increase" means that between an initial situation and the arrival one, an attribute value of an entity has changed (the size, the temperature, the position…). In fact, as we are in an algebra, we don't need to memorize any internal number, but staying at the word level is here clearly insufficient. This level of representation is named the cognitive level by (Desclés, 1990). It can be described by the same algebra, but with different kinds of operators. To achieve that, we go up to the cognitive level of the Applicative and Cognitive Grammar, and perform a deeper analysis: The attribution relation between *y* and *z* (operator ATTR in the next figure) is present at the beginning and the end. We still use the same inverter to compute the predicate at the cognitive level. The inversions for example, and in general the compositions of operators, allows to understand meaning at a semantic level, from utterances at a syntactic level; this is why, compiling high level schemes is necessary to recognize or express any utterance at lower level, as a common point between a material machine and a brain involving such a program.

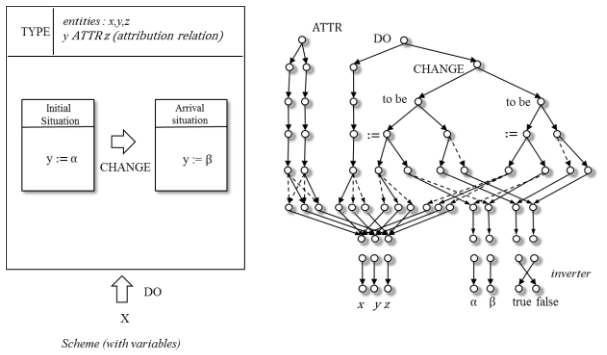


Fig. 12: Scheme of the predicate "To increase"

Conclusion

In this article, we shown compilation between isomorphic formal systems. This compilation process is generalized, not only in computer sciences or mathematics between higher and lower levels and changing representations. We carried out linguistic calculus on semantico-cognitive representations in the mathematical framework; this is a compilation between different levels using *treilles* structures. The resulting system can manage schemes (semantic) for the understanding of expressions. Examples are given in natural languages, at the meaning level; the semantic value of the predicate "go out" is the opposite of the predicate "to enter". The opposite values are connected thanks to intrication and an inversion of an application into boolean sorts. The semantic examples, as an interpretation in the linguistic field of *treilles* structures, depend of the mathematical framework. All of them are theorems and independent from material machines. A program is seen as a reading over the machine, comprehensive and potentially creating new meanings by composition of elementary meanings. This can be compiled on computers and provide an assumption of how natural language representations are operated by the brain.

References

Abraham, M. 1995. *Analyse sémantico-cognitive des verbes de mouvement et d'activité. Contribution méthodologique à la constitution d'un dictionnaire informatique des verbes*. Thèse : Paris, Ecole des Hautes Etudes en Sciences Sociales.

Aho, A., Lam, M., Sethi, R., & Ullman, J. 2007. *Compilateurs, principes, techniques et outils*. Paris: Pearson Education.

Arbib, M., & Give'on, Y. 1968. *Algebra Automata I: Parallel Programming as a Prolegomena to the Categorial Approach*. Dans Information and Control (pp. 331-345).

Backus, J. 1959. *the syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference*. Proc. Intl. Conf. Information Processing (pp. 125-132). Paris: UNESCO.

Curry, H.,Feys, R.& Craig, W. 1958. *Combinatory Logic*. Amsterdam : North-Holland.

Desclés, J.-P. 1980. *Opérateurs Opérations*. Paris: Thèse d'Etat - Université de Paris 7.

Desclés, J.-P. 1990. *Langages applicatifs, langues naturelles et cognition*. Paris: Hermès.

Desclés, J.-P. 2009. *Le concept d'opérateur en linguistique*. Histoire, Epistémologie, Langage 31/1.

Djioua, B. 2000. *Modélisation informatique d'une base de connaissances lexicales (DiSSC) : Réseaux polysémiques et Schèmes Sémantico-Cognitifs*. Thèse, Paris-Sorbonne.

Guibert, G. 2010. *Le "dialogue" homme-machine: un qui-pro-quo?* Paris: L'Harmattan.

Harris, Z. 1968. *Mathematical Structures of Language*, traduit en français par C. Fuchs. New-York: John Wiley & Sons, traduit en français par C. Fuchs, Structures mathématiques du langage, Dunod, Paris 1971.

Lawvere, W., & Rosebrugh, R. 2003. *Sets For Mathematics*. Cambridge: Cambridge University Press.

Pottier, B. 1985. *Linguistique générale, théorie et description*. Paris: Klincksieck.

Sauzay, B. 2013. *Le concept informatique de "compilation généralisée" dans les sciences cognitives (linguistique, logique et intelligence artificielle): contribution aux rapports entre la logique combinatoire et les T[Σ]-algèbres*. Thèse, Université de Paris-Sorbonne.