Intelligent Processing of an Unrestricted Text in First Order String Calculus

Andrew Gleibman

Sampletalk Technologies, POB 7141, Yokneam-Illit 20692, Israel
gleibman@sampletalk.com

Abstract. First Order String Calculus (FOSC), introduced in this paper, is a generalization of First Order Predicate Calculus (FOPC). The generalization step consists in treating the unrestricted strings, which may contain variable symbols and a nesting structure, similarly to the predicate symbols in FOPC. As a logic programming technology, FOSC, combined with a string unification algorithm and the resolution principle, eliminates the need to invent logical atoms. An important aspect of the technology is the possibility to apply a matching of the text patterns immediately in logical reasoning. In this way the semantics of a text can be defined by string examples, which only demonstrate the concepts, rather than by a previously formalized mathematical knowledge. The advantages of avoiding this previous formalization are demonstrated. We investigate the knowledge representation aspects, the algorithmic properties, the brain simulation aspects, and the application aspects of FOSC theories in comparison with those of FOPC theories. FOSC is applied as a formal basis of logic programming language Sampletalk, introduced in our earlier publications.

1 Introduction

In this work we address one of the most intriguing questions of cognitive informatics: *What kind of knowledge can be extracted from something essentially new, which is not relevant to anything known at the moment?* Simulating such cognition, we are trying to simulate the formation of substantially new knowledge in our brain. The only background knowledge, which we assume in our consideration, is the knowledge that a text has a sequential structure. We apply this assumption simultaneously for the perceived objects (that is, we are interested in perceiving and understanding the unrestricted texts) and for the means to simulate the analysis of such objects in the brain. The existing cognition and machine learning techniques usually assume less trivial frameworks. For example, the classification algorithms typically are based on geometrical properties of a feature space, which, in its turn, depends on the algorithms for extracting numerical feature values from the perceived objects.

So, we limit ourselves in a text environment and try to find the most universal features of this environment, which are independent of any particular knowledge about the text structure, the world described in the text, the language the text is written in, and the status and intentions of the writer. Still more important, we try to be independent, as much as possible, of any formal models of known phenomena. We will

M.L. Gavrilova et al. (Eds.): Trans. on Comput. Sci. V, LNCS 5540, pp. 99–127, 2009.

see that this *minimalist* approach not only leads to a simulation of the processes, occurring in the human brain, but provides a rigorous framework for building practical intelligent text processing systems.

Let us start with an analysis of the role some *artificial* symbolic notations, such as logical predicates, play in the attempts to describe and understand the external world. These notations are usually applied for expressing *known* relations between *known* objects in the context of other *known* objects, terms, relations and notations. Note that the choice of the notations and relations reflects the status of the observer rather than the inherent features of the perceived objects. This observation is crucially important for us. The cognition, based on artificial notations, assumes the usage of some previous knowledge, which may be relevant or irrelevant to the perceived world. The more sophisticated is our set of notations, the more dependent is our cognition on our previous knowledge. We will discuss some negative aspects of this knowledge and look for a way to avoid them.

A symbolic representation, which unambiguously denotes complex objects and relations, has been regarded as the main attribute of any mathematical formalization. Alonzo Church [3] describes the notations for expressing senses, or meanings, as *complex names* and assumes the following rule for manipulating them: *the sense of a complex name is not changed if we replace one of its components with another complex name, whose sense is the same as that of the replaced component.* This assumption serves as a basis for most attempts to make a flexible formal model of the external world. In the context of predicate logic the *complex names* are composed from logical terms, predicates, connectives and other symbols.

Let us look at this assumption from a specific point of view: the complexity and elaboration of the *artificially* created complex names tends to grow, making them more and more difficult for composition, combination and comprehending by a human. Over centuries such complex names are abundant with Greek letters, indices, brackets, superscripts, complex syntax, vernacular designations etc. Usually every object and relation, subjected to formalization or modeling on a computer, is notated with a specially devised artificial symbolic notation. The notations often need to be accompanied with informal comments or substantial documents, which help understand the meaning of the involved symbols.

Do we always need so elaborated *artificial* symbolic notation to abide to Church's assumption mentioned above? Can we apply, instead of the artificial notations, the notations which already exist in *natural* sources, such as unrestricted natural language texts or biological sequences? Do we always need to apply a *prior* semantic knowledge, expressed mathematically, in order to make inferences from such data?

Interestingly, we can formalize our knowledge and define algorithms without complex artificial syntax at all. Known text examples, such as encyclopedia articles and patent formulations, show that we can describe very complex things without explicitly defined formal notations and appeal only to the common meaning of natural language words and expressions. Can we do this in a formal language? Can we apply *natural* phrases and their senses in algorithm definition? Can we use natural language words and concepts in a computer program as fluently as in human speech, without having to *explicitly* define the artificial *complex names* for that? Can we extract and apply patterns of unrestricted texts immediately in reasoning, consistently with the meaning of these texts? Is there a possibility to define what this meaning is without reference to any mathematical constructions known a priori?

We will clarify these questions by challenging the concept of logical predicate. For this purpose we develop a special kind of logic, which can be termed *a predicateless logic* or *a string calculus*. Using this logic, we can compose a *formal* theory or an algorithm using *natural* words and phrases that do not have explicit formal definitions.

On the first glance this seems impossible. Note, however, that in our speech we can correctly apply numerous words without knowing the *explicit* definition of their senses. We met these words in the phrases, which we understand, and we are able to apply these words in a similar context. We have learned them *implicitly*, via the contexts. Can we do this on a computer? Can we apply unrestricted words, which are not defined explicitly, in a formal definition that a computer may understand, consistently with the common meaning of these words? If we can do so, we can *immediately* employ the meaning of numerous natural language words and phrases, existing in such knowledge sources as work instructions, encyclopedia articles and patent formulations, in order to make inferences from this knowledge.

This work should be considered an attempt to do this in a Logic Programming framework. Logic Programming systems typically are based on the following two fundamental principles: *a logic term unification principle* and *a resolution principle*. The first principle allows manipulation with senses, according to Church's assumption mentioned above, by unification of logical terms: *term constituents, placed in similar positions of the unified terms, have similar senses*. The second principle provides a reasoning engine for deriving logical conclusions from logical assumptions. In this work we make a change in the first of these principles. Instead of the logical terms we apply unrestricted strings, which may contain variables and some nesting structure. We define and control senses by a unification of such strings, so that *string constituents, placed in similar positions of the unified strings, have similar senses.*

So, we compose logic clauses from the unrestricted strings, which may contain variable symbols and a nesting structure. We apply the resolution principle to such clauses in the same way it is applied in reasoning systems based on first order predicate calculi. Keeping an analogy with First Order Predicate Calculus (FOPC) terminology, we call the corresponding calculus First Order String Calculus (FOSC). FOSC is equivalent to FOPC when the class of strings, used in the clauses, is reduced to well-formed string representations of logical atoms.

The main advantage of this approach to logic is a complete elimination of logical terms and predicates from logical reasoning. Indeed, the need to design the artificial logical terms and predicates has obvious disadvantages. It is a complex intellectual work. The designed predicates may make the system biased and disregard important data features. Using FOSC, we can apply the classical logical reasoning methods without any artificial logical predicates at all. Instead of the predicates we use the patterns of unrestricted strings, which can be taken immediately from natural domain texts.

The patterns are produced using *alignment*, *generalization* and *structurization* of the original text fragments. They define the necessary formalization according to the regularities, applied via string *unification*. In this way we can produce an unbiased analysis of a text of unrestricted nature.

This work is related to cognitive informatics as follows. Jean Piaget [20] and his successors recognize that small children can generate and apply some kind of symbolic representation of the surrounding world and can expose a logically reasonable behavior. However, the children do not use explicit predicate notations. In order to make conclusions and conjectures, a child can apply *subconscious* versions of modus

ponens and Church's substitution rule mentioned above. The child generally operates with *implicit* concepts, learned by examples, rather than with explicit concept definitions. This is a principal contradiction between the child's mind and the development of mathematics. In order to implement in our computer what a child can do easily and subconsciously, today we need a whole industry of manipulation with complex artificial notations! Consider formal and programming languages. Church's assumption is generally abided to, but the abundance of complex notations for explicit definitions often makes the relations between the algorithm components and their meanings hardly tractable.

The paradox we see here is that, when building computerized models, people tend to apply the *explicit* methods even where this leads to cumbersome and expensive models of simple phenomena. In order to understand this paradox, we try to understand what kind of formal elements *already exist* in unrestricted *natural* sources, such as natural language phrases and, in a future perspective, unknown codes, biologic sequences, images and other patterns of natural phenomena.

Exactly this objective leads us to the attempts to extract the regularities, contained in the raw domain texts, via the alignment and generalization of similar text fragments. FOSC can be considered *a knowledge representation tool, where all the knowledge is encoded using the patterns of raw text fragments treated as demonstrating examples*. The knowledge modeling and manipulation remains logic-based. The formal inference is based on the knowledge, extracted from similar text examples by alignment and generalization. In this way the *data* is transformed into *knowledge*. This can be considered a modeling of the internal knowledge in the brain. Indeed, in our brain we relentlessly align, match, compare, generalize and compose the images of external objects, contained in our memory. We still do not know how we do this. In FOSC we are trying to simulate these processes using the unrestricted text patterns, as described in Sections 2 and 3.

Generalization universe, introduced in Section 2, provides a special mathematical mechanism for extracting the knowledge, contained in a set of unrestricted strings, independently of any other knowledge. Although the introduced pair-wise string generalization and unification operations are not deterministic, we can consider a generalization universe as algebra with such operations. It can be considered a kind of concept algebra, where the concepts are *demonstrated* in string examples, but not formulated explicitly. Although we still do not specify the pair-wise string generalization algorithm (this can be done in many ways), the concept of generalization universe allows one to understand the expressive potential of the knowledge contained in the unrestricted texts. Indeed, we show (Section 3) that *any* algorithm can be encoded by alignment of suitable string examples. In this way we can study the inherent algorithmic content of a text independently of any other knowledge, and assess its cognitive complexity.

This is essentially an autonomic knowledge processing. The independence from other knowledge sources enables us to apply the inherent meaning of the unrestricted texts in a totally autonomic fashion. Furthermore, we can compose such meanings for the understanding of more complex texts, keeping the independence from any knowledge created by other means. This is discussed in Sections 4 and 5 (see also paper [6]), where we describe our experiments with FOSC, and in the concluding sections of the paper.

Some details of our experimental implementation of FOSC are described in Section 6 and in the Appendix.

2 Basic Definitions of First Order String Calculus

Let T be an infinite set of *terminal symbols*, V be an infinite set of *variable symbols, or variables*, $T \cap V = \varnothing$, and special symbols ¬ (not-sign), [and] (square brackets) do not belong to the sets T and V.

Finite strings, consisting of terminal symbols, are called *unstructured ground strings*. Unstructured ground strings and the results of operations of string structurization and string generalization, described below, are called *strings*. A *segment* of string S is a fragment of string S, which starts at some index i and ends at some index j, $i \le j$.

Definition 2.1. A process and a result of inserting special square brackets [and] (in this order) into a string S are called *a structurization* of string S. That is, if S has a form $\alpha s \beta$, where α, s, β are segments, then $\alpha[s]\beta$ is the result of structurization. Here we assume that if segment s already contains a square bracket, then this segment contains the second bracket of the bracket pair, i.e. s is a valid string concerning its square bracket structure. Segments α and/or β may be empty.

Definition 2.2. A process and a result of substituting non-intersecting segments in a string S with variables are called *a generalization* of string S. Like in the previous definition, we assume that the substituted segments are valid strings. Also we assume that multiple occurrences of a single variable, introduced in a generalization of string S, are corresponding to equal segments of string S. The latest assumption is important for what follows; it is analogous to the assumption of uniqueness of values of mathematical functions. See examples at the end of this section.

Notation $f:S_1 \rightarrow S_2$ will stand for a generalization of string S_1 into string S_2. A sequence of segments of string S_1 (counted from left to right), which are substituted in the generalization, is called *a kernel* of the string generalization. A sequence of segments of string S_2, separated by the variables introduced in the generalization, is called *an image* of the string generalization. A generalization $f:S_1 \rightarrow S_2$ is called *reversible* if the corresponding backward substitution $f^{-1}:S_2 \rightarrow S_1$ is also a generalization; otherwise it is called *irreversible*, or *degenerate*. The result of sequential generalizations $f_1:S_1 \rightarrow S_2$ and $f_2:S_2 \rightarrow S_3$ is designated as $f_1f_2:S_1 \rightarrow S_3$ or $f_1f_2:S_1 \rightarrow S_2 \rightarrow S_3$.

Definition 2.3. A pair of string generalizations $\{f_1:S_1 \rightarrow S, f_2:S_2 \rightarrow S\}$ with the same image and the same variables, used for the substitution, defines *an isomorphism* between strings S_1 and S_2. (Here we assume that strings S_1 and S_2 do not contain common variables). String S is called *a common generalization* of strings S_1 and S_2. Segments of S_1 and S_2, related to the same variable in S, are called *synonyms* with respect to the string isomorphism. Synonyms take similar places in so aligned strings (see the examples at the end of this section). So defined string isomorphism is called *most specific* if it can not be represented in the form $\{f_1h:S_1 \rightarrow S' \rightarrow S, f_2h:S_2 \rightarrow S' \rightarrow S\}$, where $h:S' \rightarrow S$ is a degenerate generalization. In this case string S is called *a most specific common generalization* of strings S_1 and S_2. See examples in the final parts of this section and in Section 4.

Operation of string generalization, considered a relation $g(s_1, s_2)$: "string s_2 is a generalization of string s_1", forms a preorder on the set of all strings defined above. The maximal elements in this set are strings, formed by single variables. The minimal elements are ground strings.

Definition 2.4. The result of simultaneous substitution of all occurrences of a variable in a string S with some string is called *an instantiation* of string S. We assume that such substitution can be done simultaneously for one or more variables of the string S.

String generalization and instantiation are inverse relations of each other. Strings S and S′ are called *equivalent* to each other if they are instantiations of each other. String equivalence should not be confused with string isomorphism, in which the strings are aligned in order to produce a common generalization.

String equivalence forms a relation of equivalence on the set of all defined strings. Equivalent strings can be transformed to each other by a reversible generalization. Speaking of strings with variables we usually bear in mind the corresponding equivalence classes. Relation of preorder $g(s_1, s_2)$ between strings becomes a relation of partial order between the classes of equivalent strings.

It can be easily shown that any string generalization can be represented as a most specific common generalization of a pair of ground strings, which have some similarity. The non-matching fragments of so aligned strings become synonyms with respect to the corresponding string isomorphism. (See examples in the final part of Section 4).

Definition 2.5. Given a set E of ground strings, its closure with respect to the operation of taking most specific common generalization, factorized by the relation of string equivalence, is called *a generalization universe* G(E) of set E. (See example in Fig. 1 and in the table that follows). The elements of generalization universe G(E) correspond to all possible regularities, which can be discovered by multiple pair-wise alignments of elements of set E. More specific regularities correspond to the lower elements in the ordering of string generalization. This concept of regularities is discussed in more detail below.

Definition 2.6. Given two strings S_1 and S_2, consider a possibility to create a common instantiation S of these strings. String S and the corresponding substitution of variables (if possible) are called *a unification* of strings S_1 and S_2.

Definition 2.7. For any string S, *a string literal* (or simply a literal), produced from S, is defined either as S or ¬S. (Remind that not-sign ¬ does not belong to the sets of symbols T and V). Keeping an analogy with mathematical logic terminology, we call S and ¬S *a positive* and *a negative* literal respectively.

Definition 2.8. A pair (H, B), where H is a string and B is a finite set of string literals, is called *a string clause,* or simply a clause. Set B can be empty. Keeping an analogy with Horn clauses, considered in Logic Programming, we call H and B *a clause head* and *a clause body* respectively. String clause (H, B) is called positive if its body B contains only positive literals.

A notation H :– B will be applied for the designation of a string clause with head H and non-empty body B. A clause with head H and empty body will be designated simply as H. A unification of clause head H of a string clause H :– B with string S of a literal S or ¬S is called *an instantiation of the string clause* by this literal. We assume that the substitution of the corresponding variables is done globally in the entire clause H :– B, including its body.

Definition 2.9. A finite set of string clauses is called *a string theory, or FOSC theory*. We say that a string theory P is *composed* of strings, applied in its literals.

Definition 2.10. Assume that an instantiation of a positive string clause (H, B) by a ground string α makes every string $S \in B$ unifiable with some string $H_{\alpha,S}$. Clause (H, B) is called *a description clause* for the ground string α. String α is called *the supporting example;* string H is called *bondable* by the set B; elements of set B are called the *bounds*; string H is also called *the acceptor* of the bounding; string $H_{\alpha,S}$ is called the attractor for bound S.

Examples. In our examples we apply the following notation. Variable names and terminal symbols in strings are separated by a white space and/or square brackets. Variable names start with capital Roman letters and may contain only Roman letters and digits. The terminal symbols are designated by sequences of characters and letters (excluding white space, the neck-symbol :–, double dots, double commas and characters ¬, [,]) and do not start with capital Roman letters. Like in Prolog language, proper names, considered as terminal symbols, are written non-capitalized. Strings in a clause body are separated by double commas and are terminated by a double dot at the end of the clause. Program comments start with the percent sign %.

Consider strings S_1, S_2, S, S′, defined as follows:

S_1 = "[a , b] ⇒ [b , a]", S = "[A , B] ⇒ [C , A]",
S_2 = "[x , y , z] ⇒ [z , y , x]", S′ = "[A , B] ⇒ [B , C]".

These strings contain distinct terminal symbols a, b, x, y, z, comma, ⇒, and distinct variable symbols A, B, C. Remind that square brackets [and] are special symbols, which define a nesting structure on strings. Strings S_1 and S_2 demonstrate a reversing of comma-separated lists of entities. String S realizes a regularity, which may be verbally formulated as follows: the first element of the first comma-separated list in square brackets is equal to the last element of the second comma-separated list in square brackets. String S′ realizes a slightly different regularity.

A generalization $f:S_1 \to S$ is defined by substituting a with A, 1st occurrence of b with B, 2nd occurrence of b with C. A generalization $g:S_2 \to S$ is defined by substituting x with A, "y , z" with B, "z , y" with C. Pair {f, g} defines an isomorphism between strings S_1 and S_2. String S is a common generalization of strings S_1 and S_2.

A generalization $f':S_1 \to S'$ is defined by substituting 1st occurrence of a with A, b with B, 2nd occurrence of a with C. A generalization $g':S_2 \to S'$ is defined by substituting "x , y" with A, z with B, "y , x" with C. Pair {f′, g′} defines another isomorphism for strings S_1 and S_2. String S′ is another common generalization of strings S_1 and S_2.

Strings S and S′ are not equivalent, so we find that a pair of strings may be generalized into two essentially different, i.e. non-equivalent and not generalizing one another, common generalizations.

The following string alignments define two essentially different unifications of string S with string [c a t , d o g , m o n k e y] ⇒ [W]:

[	A	,	B	] ⇒ [	C , A	]
[	c a t	,	d o g , m o n k e y	] ⇒ [	W	]

[	A	,	B	] ⇒ [	C , A	]
[	c a t , d o g	,	m o n k e y	] ⇒ [	W	]

The alignments differ in the matching of commas. The unifications are defined by the substitution of variables A, B and W:

[c a t , d o g , m o n k e y] ⇒ [C , c a t] (from the 1st alignment),
[c a t , d o g , m o n k e y] ⇒ [C , c a t , d o g] (from the 2nd alignment).

The following strings are instantiations of the first unification:

[c a t , d o g , m o n k e y] ⇒ [m o n k e y , d o g , c a t],
[c a t , d o g , m o n k e y] ⇒ [j a b b e r w o c k y , c a t] .

The following strings are instantiations of the second unification:

[c a t , d o g , m o n k e y] ⇒ [m o n k e y , c a t , d o g],
[c a t , d o g , m o n k e y] ⇒ [j a b b e r w o c k y , c a t , d o g].

In Fig. 1 a fragment of the generalization universe, created from a set of ground strings containing strings S_1 and S_2, is depicted. It contains strings S and S′. Note that the depicted strings are representatives of the corresponding equivalence classes. For instance, the class of equivalent strings, which is represented by string [A , B] ⇒ [C , A], contains string [X , Y] ⇒ [Z , X], where X, Y, Z are distinct variables. The less general elements of the generalization universe realize more specific regularities.

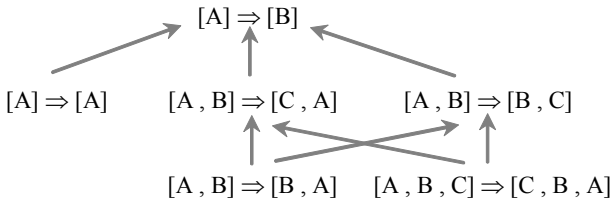

Fig. 1. A fragment of generalization universe of the set of ground strings "[a] ⇒ [a]", "[b] ⇒ [b]", "[a, b] ⇒ [b, a]", "[x, y] ⇒ [y, x]", "[a, b, c] ⇒ [c, b, a]", "[x, y, z] ⇒ [z, y, x]". The arrows show the relation of string generalization.

Below is a simple string theory example:

Example 2.1. A string theory of reversing comma-separated lists

[A , B] ⇒ [C , A] :– [B] ⇒ [C]..
[A] ⇒ [A]..

This is a theory of reversing comma-separated lists of lists of terminal symbols. The application of a string theory to a string is defined in the following section (see Algorithm 3.1). Remind that ⇒ is an ordinary terminal symbol, and that string clauses in a string theory are terminated by a double-dot.

The application of the above theory to the string [c a t , d o g , m o n k e y] ⇒ [W] will produce the string [c a t , d o g , m o n k e y] ⇒ [m o n k e y , d o g , c a t]. This is done recursively as follows. The input string (goal) is unified with the clause head [A, B] ⇒ [C, A], forming a new sub-goal [d o g , m o n k e y] ⇒ [C] from the clause

body [B] ⇒ [C]. In the next recursion step, a sub-goal [m o n k e y] ⇒ [C] is similarly formed. This sub-goal can be unified only with the clause [A] ⇒ [A], resulting in string [m o n k e y] ⇒ [m o n k e y]. The output string is then formed by the variable substitutions on all recursion levels. Note that in this theory the 2nd possibility of aligning text [A, B] ⇒ [C, A] with the goal is not applied.

This example demonstrates an incorporation of parsing facilities immediately into a logical reasoning, *avoiding the concepts of predicate and formal grammar*. Some very important implications of this avoiding are discussed in Section 4 in relation to natural language syntax and semantics.

Consider Definition 2.10. The first clause of Example 2.1 is a description clause of ground example [a , b] ⇒ [b , a]. String [A , B] ⇒ [C , A] is the acceptor of the corresponding bounding; string [A] ⇒ [A] is the attractor of the bound [B] ⇒ [C].

Now, the same clause is a description clause of ground example [a, b, c] ⇒ [c, b, a]. String [A, B] ⇒ [C, A] is simultaneously the acceptor and attractor of this bounding. Our terminology of acceptors and attractors will be clarified in Section 7.

In the following example, a fragment of generalization universe G(E) of a set of English sentences is shown. Sentence numbers in the columns mark the sources for building most specific string generalizations. The most general of these generalizations is underlined near the bottom of the table.

Sentence No.	*Set of English sentences*	*Sentence No.*	*Generalizations derived from 2 sentences*
1	the crayon is found by mary	1,2	the X is found by mary
2	the pencil is found by mary	1,3	the X is Y by mary
3	the book is stolen by mary	1,5	the X is found by Y
4	the book is returned by mary	3,4	the book is X by mary
5	the book is found by bill	3,5	the book is X by Y
6	the book is found by john	5,6	the book is found by X

Sentence No.	*Generalizations derived from 3 or more sentences*
(1,2),(1,3)	the X is Y by mary
(1,2),(1,5)	the X is found by Y
(1,2),(3,5)	<u>the X is Y by Z</u>
(3,4),(5,6)	the book is X by Y

3 Algorithmic Form of a FOSC Theory

We introduced FOSC, abandoning the concept of predicate and trying to keep other characteristics of FOPC, related to logic programming. Now we are going to apply FOSC as a means for creating algorithms. *From now on we consider the string theories and the bodies of string clauses as sequences*. Any Prolog program can be considered a string theory.

A string theory defines an algorithm for string processing as described below. The algorithm is defined recursively, keeping an analogy with the application of a Prolog program P to a Prolog goal S. String unification algorithm, applied here, is a simple modification of the common logic term unification algorithm (see Robinson, [22];

Lloyd, [15]), which is specified for working with the unrestricted strings with a nesting structure and variables. The details of this algorithm are not important for the understanding of what follows; interested readers can find them in the Appendix.

Algorithm 3.1. Apply a string theory P to a string-goal S.

1) Starting from the top of P*, find the first clause* H :– B*, not considered before, such that* H *can be unified with* S*. Return an unsuccessful status and exit if this is impossible.*
2) Build a clause H_1 :– B_1 *as the first possible instantiation of clause* H :– B *by string* S*, not considered before. Proceed to Step 1 if this is impossible (that is, all such instantiations are already considered).*
3) If B_1 *is empty then return* H_1 *as the successful result of the algorithm and exit. Otherwise sequentially apply* P *to the strings of all positive and negative literals of* B_1*. In case of success, return* H_1 *as the successful result of the algorithm and exit. In case of non-success proceed to Step 2.*

In Step 3 we assume that string H_1 is modified by global substitutions of its variables (if any), made during the corresponding unifications. Also we assume that the application of P to any negative literal returns an unsuccessful status. The latest assumption is related to the closed-world assumption (CWA), which states that what cannot be deduced from a theory is considered wrong. The application of P to a negative literal is attempted only if the string of this literal is a ground string (this is so called safeness condition). This algorithm is based on SLDNF-resolution rule; see [15].

It is now the best time to consider Example 4.1, where we analyze the application of a natural language related string theory to natural language related goals.

The representation of algorithms as string theories is Turing-complete, which means that we can represent any algorithm as a string theory. In order to prove this we will prove that any normal Markov algorithm [17] can be represented as a string theory. This is stated in the following theorem.

Theorem 3.1. Let M be any Markov algorithm, written in a base alphabet B and a local alphabet L, where $(B \cup L) \subset T$ and T is the set of terminal symbols applied in string theories. There exists a string theory P_M, which, given a goal $[\xi t \xi] \Rightarrow [W]$ (where t is a string, ξ and $\Rightarrow$ are terminal symbols not belonging to alphabets B and L, W is a variable), does the following:

- *Transforms the goal into a string* $[\xi t \xi] \Rightarrow [\xi t' \xi]$ *and stops if* M *transforms string t into a string t′ and stops;*
- *Produces a string* $[\xi t' \xi] \Rightarrow$ [[fail]] *and stops if no rule of* M *is applicable to some derivation t′ of t, produced by* M*;*
- *Never stops if string t leads* M *to an infinite application of the Markov rules.*

The theorem can be proved simply by rewriting the rules of Markov algorithm $\alpha \rightarrow \beta$ (normal rules) and $\alpha \rightarrow . \beta$ (stop rules) in the form of FOSC clauses $[X\alpha Y] \Rightarrow [W]$:– $[X\beta Y] \Rightarrow [W]$ and $[X\alpha Y] \Rightarrow [X\beta Y]$ (a clause with empty body) respectively. Here *X*, *Y* and *W* are FOSC variables. Clause $[X] \Rightarrow$ [[fail]] (also a clause with empty body) is added at the end of the theory in order to prevent backtracking.

Representation of Markov algorithms, written in other alphabets, can be done by encoding, using the set T of terminal symbols. So, for any Markov algorithm we can construct a FOSC theory with a similar behavior. Now, any Turing machine can be represented as a Markov algorithm [17], so we can conclude that any Turing machine can be represented as a string theory and that the possibility of representing algorithms in the form of string theories is also Turing-complete. The same result can be proven in a different way, by observing that Algorithm 3.1 subsumes context-sensitive string rewriting.

Theorem 3.1, along with the means for pair-wise string generalization and unification, described above, leads to an interesting treatment of Church-Turing thesis. For a detailed description of this thesis, see e.g., [12]. In a common formulation the thesis states that every effective computation can be carried out by a Turing machine. The thesis contains the following requirements about an algorithm (or a method):

1). The method consists of a finite set of simple and precise instructions, which are described with a finite number of symbols.
2). The method will always produce the result in a finite number of steps.
3). A human being with only paper and pencil can in principle carry out the method.
4). The execution of the method requires no intelligence of the human being except that which is needed to understand and execute the instructions.

Instructions, mentioned in p.1, usually are organized in a form of formal algorithm description. Some representation of the input data for the algorithm is also assumed.

String theories provide an alternative view. We do not need the *instructions*, mentioned in p.1. Instead, we use a finite set of *data examples* (in a string form), produce the generalized patterns of these examples, and combine the patterns using a universal set of rules, which are described in Algorithm 3.1.

Data example patterns now become a fundamental part of the algorithm. So we came to the following interpretation of Church-Turing thesis:

Thesis 3.1. *Every effective computation can be carried out using sequences of generalized patterns of data examples and a universal set of rules for combining such sequences.*

Indeed, any string theory consists of clauses, each of which is a sequence of generalized string examples. The universal set of rules for combining such sequences is defined in Algorithm 3.1.

The set of instructions now consists of the following two distinct subsets: a) a fixed set of universal rules, determining how to apply a string theory to a string; b) a finite set of data generalizations, which now play the role of instructions, or model examples, for the processing of another data. We especially emphasize that the set of *actions* (instructions of type (a)) here is fixed forever. All other means for creating algorithms are essentially *data*, which can be extracted from the unrestricted text fragments by a generalization.

Now, any such data generalization can be derived from a pair of demonstrating examples, which have some similarity, using a most specific common generalization. So we come to the following additional interpretation of Church-Turing thesis, where the process of generalization itself is a part of the computation:

Thesis 3.2. Every effective computation can be carried out using sequences of data example pairs and a universal set of rules for combining such sequences.

In more general terms we can reformulate both theses as follows:

Thesis 3.3. Every possible computation can be carried out by alignment and matching of some data examples.

As an important implication of this thesis, we can build any algorithm without specially designed instructions for the definition of actions, logic conditions, relations, data structures, properties, procedures etc. Everything is *automatically* defined by the regularities, contained in the generalized string examples. Likewise, we can build a formal theory of any data, presented in a string form, without any previously defined mathematical concepts, grammars, or predicates related to the data. Every concept is *implicitly* defined by the corresponding variable, which represents the non-matching string constituents aligned for producing the string generalization. In Sections 4 and 5 we provide examples and discuss this implicit definition of concepts in more detail.

When we *think* of an algorithm, we usually think of steps, which are applicable to similar data examples. Now we see that we can *define* algorithms and theories, using only data examples, which have some similarity. Any known algorithm or formal theory has a prehistory of finding similarities in data, which may become a part of this algorithm or theory if we reformulate it according to Thesis 3.3.

This observation is consistent with our intentions to simulate the internal knowledge processing in the brain. According to our hypothesis, described in the introduction section, the alignment, matching, generalization and composition of string examples into clauses in FOSC simulate the way our brain manipulates with the images contained in our memory.

4 Case Study: Implicit Syntax and Semantics of Natural Language Text as a FOSC Theory

Recall once more the assumption by Alonzo Church, mentioned in our introduction. In Church's context the mathematicians manually compose the *complex names*. Every applied predicate and functional symbol should be explicitly declared and defined. The senses are defined *explicitly*.

Instead of doing this, we can align and generalize sample strings, where the senses are *demonstrated*, and then apply these senses using the generalized string patterns. Text constituents, related to each other in the string alignment, define the senses *implicitly*. It is important to realize that we extract and operate with these senses without the explicit formal definitions. We just *align* some sample strings, *assume* that this alignment is correct, make the generalization, and then use so introduced variables in clauses via clause instantiation. Like explicit formal definitions, such an implicit definition of senses can be correct (if our assumptions about the string alignments are sound) or incorrect (if they lead to a wrong inference). Clauses, composed of the generalized strings, serve the role of axioms for deriving the conclusions.

Let us apply this framework for the definition and manipulation of senses of natural language phrases. A FOSC theory can be considered as ontology in the following way:

- Word occurrences in clause heads are considered the concept definitions;
- Word occurrences in clause bodies are considered the suggested concept applications;
- Semantics of the words and word combinations is defined by the suggested concept applications.

Consider the problem of question answering. Assume that there exists a set of simple English sentences, not limited by any specific grammar, which describe certain facts. (See the facts in Example 4.1; the structurization of these sentences will be discussed later). The facts describe a simple situation, which may happen in a real-world library or store.

Note that we *intentionally* do not apply any linguistic concepts: everything will be defined implicitly. We would like to develop a question answering system, which answers questions about the facts. Also we would like our system to be easily expandable by new facts, presented just in a simple English, and new kinds of questions.

Some common approaches for formalizing the involved NLP phenomena are based on abstruse explicit formalisms, which include formal grammars, first order logic with numerous predicates, graph-based representations of semantics, large thesauri with semantic tags etc. Using FOSC we can simplify this. In Fig 2 we depict the difference between the application of FOSC to this NLP problem (see the white rectangles only) and the approach based on FOPC combined with a formal grammar formalism (see *all* the rectangles, assuming a predicate-logic form of the inference engine, ontology and the formal model).

We now discuss two implementations of the same algorithm for question answering, which are created according to the paths described in Fig. 2: path 1-2 and path 3-4-5-6-7-8. The FOSC implementation (path 1-2) is given in full details. The FOPC implementation, which is much more complex, is described briefly.

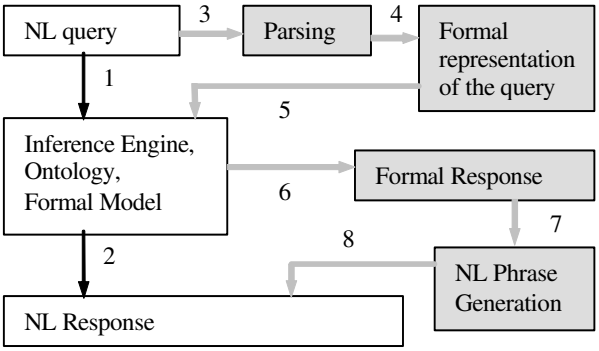

Fig. 2. Comparison of FOPC and FOSC approaches to question answering: ⟶ *FOPC approach* assumes the application of formal grammars and leads to the need to design predicates, grammars, a semantic model, and implement a parser, a formal reasoning engine, and a NL phrase generator. ⟶ *FOSC approach:* Everything is defined via the immediate interaction of generalized patterns of NL phrase examples.

FOSC theory, presented in Example 4.1, forms the answers according to the meaning of natural language phrase patterns, contained in its clauses. In this example we intentionally combine a linguistic type of inference with a logic inference, related to the subject. The theory contains a set of inference rules, followed by some facts in a simple English form. Note the negation signs ¬ in the 5th and 8th clauses. Note also that the scope of any variable is limited by the clause containing this variable.

Example 4.1. A FOSC theory for question answering.

```
can [A] see [X] ? :– [A] can see [X]..                    % Inference rules
what can [A] see ? [X] :– [A] can see [X]..
what is in [A] ? [X] :– [X] is in [A]..
what is visible on [A] ? [X] :– [X] is visible on [A]..
what is invisible on [A] ? [X] :– [X] is on [A] ,, ¬[X] is visible on [A]..
what is on [A] ? [X] :– [X] is on [A]..

[A] can see [X] :– [A] is standing near [B] ,, [X] is visible on [B]..
[A] is visible on [B] :– [A] is on [B] ,, ¬[A] is in [closed X]..
[A] is standing near [B] :– [A] has approached [B]..
[X] is on [A] :– [X] is in [B] ,, [B] is on [A]..

[a book] is in [open box]..                               % Facts
[a notebook] is in [closed box]..
[open box] is on [red table]..
[closed box] is on [red table]..
[john] has approached [red table]..
```

Consider the following input queries for the theory:

1) can [john] see [a book] ?
2) can [john] see [a notebook] ?
3) can [john] see [open box] ?
4) what can [john] see ? [X]
5) what is visible on [red table] ? [X]
6) what is invisible on [red table] ? [X]
7) what is on [red table] ? [X]
8) what is in [open box] ? [X]
9) what is in [closed box] ? [X]

Variable X stands for the fragments of the query results, which will be instantiated during the inference. The application of the theory to the above inputs provides the following results respectively:

1) can [john] see [a book] ? *Yes*.
2) can [john] see [a notebook] ? *No*.
3) can [john] see [open box] ? *Yes*.
4) what can [john] see ? [open box]. *Yes*;
   what can [john] see ? [closed box]. *Yes*;
   what can [john] see ? [a book]. *Yes*.
5) what is visible on [red table] ? [open box]. *Yes*;
   what is visible on [red table] ? [closed box]. *Yes*;
   what is visible on [red table] ? [a book]. *Yes*.

6) what is invisible on [red table] ? [a notebook]. *Yes*.
7) what is on [red table] ? [open box]. *Yes*;
   what is on [red table] ? [closed box]. *Yes*;
   what is on [red table] ? [a book]. *Yes*;
   what is on [red table] ? [a notebook]. *Yes*.
8) what is in [open box] ? [a book]. *Yes*.
9) what is in [closed box] ? [a notebook]. *Yes*.

Words *Yes* and *No* in Italic here indicate a successful or unsuccessful status of the application of the theory to the query. Some queries have multiple answers. The negation in query (2) is done according to the closed-world assumption rule. Remind that square brackets define nesting structures on the applied string patterns and constrain the unification.

Essentially, this theory contains some algorithmic knowledge about syntax and semantics of English words and phrases, and allows us to apply this knowledge. Curiously, we did not try to define this algorithmic knowledge explicitly, using previously formalized artificial lexical, syntactical or semantic concepts. *We apply this algorithmic and linguistic knowledge immediately by examples!*

The theory expresses and supports reasoning about common relations between objects (a book, a notebook, john etc). Among the relations we find *inclusion relations* "is in", *topological relations* "is on", "is standing near", *time-related relations* and *action patterns* "[A] has approached [B]" and relations concerning a visibility of objects and an ability of a person to perceive objects. What is important for us, the relations are defined *implicitly*, via the generalized examples of their usage. The roles and types of the objects-operands of the relations are also defined implicitly, according to the word positions in the phrase patterns.

We could define the relations *explicitly* through a conventional artificial FOPC form (e.g., a Prolog form) by inventing the following predicate notations:

```
is_in (a_book, open_box).
is_in (a_notebook,  closed_box).
is_on (open_box, red_table).
is_on (closed_box, red_table).
has_approached (john, red_table).
```

If we try this, we will find that we need to invent additional explicit notations for the relations and to create the corresponding axioms, e.g.

```
can_see (A, X) :– is_standing_near (A, B), is_visible_on (X, B).
is_visible_on (A, B) :– is_on (A, B).
is_standing_near (A, B) :– has_approached (A, B).
```

Such predicates and axioms would define the FOPC semantics according to arrows 5 and 6 in Fig. 2. However, we will also have to build a grammar for parsing the input queries 1-9 and for transforming them into the predicate form (see arrows 3 and 4). Then we will have to implement the corresponding parser. Then we will need another grammar and a special phrase generator for generating a natural language representation of the inference results (see arrows 7, 8).

Trying to develop this FOPC-based formalization, we will find that the simplicity, readability, independence from other formalisms, and the flexibility for developing the theory are lost forever. This is a confirmation of our complexity observation, related to the assumption by Alonzo Church, given in the introduction section.

Using FOSC we can avoid these complications (see arrows 1, 2 in Fig. 2): in our theory the artificial concepts of formal grammar and logical predicate are not applied at all. So, FOSC provides an alternative mechanism for the formalization of text structure and meaning, as well as for text analysis and generation.

Although we avoid the predicate notations, our theory is still *a formal theory*, conforms to Church's assumption, and supports a logical reasoning, which can be applied to other objects, phrase patterns and lexemes. We not only exploit the *syntax* of the involved phrases. We exploit the inherent *semantics* of the English words and word combinations "can see", "is in", "is visible", "has approached" etc, without having to create the explicit formalization of this semantics. In this way we can take advantage of the syntax and semantics of any natural text fragment without the need to define them explicitly.

String patterns, applied in Example 4.1, can be created as follows. The following table contains a verbal description of the meaning of several string pairs, composed of similar English sentences and dialogue samples:

String example pair	*Verbal description*
can john see a book ? can mary see open box ?	Question about the visibility of some object or about the ability of a person to perceive objects
john can see red table mary can see closed box	Statement that some object is visible or that some person can perceive objects
what can john see ? a book what can mary see ? open box	Question and answer about the visibility of some object and about the ability of a person to see this object
what is in open box ? a book what is in room ? red table	Question and answer about a containment

When we have such string pairs, we do not need to *write* a theory. We can *build* it, manually or automatically, from the string pairs, using structurization and most specific common generalization (cf. Section 2) as follows:

String example pair	*Structured most specific common generalization*
can john see a book ? can mary see open box ?	can [A] see [X] ?
john can see red table mary can see closed box	[A] can see [X]
what can john see ? a book what can mary see ? open box	what can [A] see? [X]
what is in open box ? a book what is in room ? red table	what is in [A] ? [X]

The *implicit definition* of the semantics assumes such an abduction of the semantics from the natural language text samples, where this semantics is *demonstrated*, rather than an explicit mathematical definition what this semantics is. In terms of the assumption by Alonzo Church, we create the *complex names* for a logic formalization of a problem immediately from the generalized patterns of natural texts, belonging to the problem domain, rather than from the artificially constructed logical atoms and predicates. We create an algorithm *by incorporating the demo examples immediately into the algorithm description*, according to Thesis 3.1.

Example 4.1 is a "toy" reasoning system, which only demonstrates that we can do without the *explicit* frameworks, such as formal grammars and predicates. In this work we only show that FOSC can be applied as an alternative to these techniques, enabling us to avoid some of their problems. We did not discuss the automatic acquisition of inference rules for specific NLP tasks: we relate this to a future research and experiments. Note that some modern corpora and machine learning methods for building large grammars and transfer bases can be similarly applied for building FOSC rules.

Some problems of ambiguity and variability in natural languages can be tackled by the acquisition and alignment of phrases, which have similar structure and meaning, as shown in our examples. As an interesting example of avoiding ambiguity, consider the word "can", contained in our examples. Example 4.1 implements reasoning, related to a treatment of this word as a modal verb. Due to the constraints, imposed by unification of the phrase patterns, containing this word, this reasoning cannot be applied when this word is correctly treated as a noun. So*, the ambiguity is simply avoided, just like we avoid it in our speec*h. We did not exert any efforts for resolving this ambiguity besides the application of correct phrase patterns: the ambiguity simply does not arise.

Now let us recall the questions, asked in the introduction section. Our examples show the following.

Conclusions from the case study. We can abide to Church's assumption without *artificial* symbolic notations: we apply the notations, which already exist in *natural* sources. We only need to make the generalization and structurization of the natural strings, such as natural language phrases, suitable for our purposes.

We can apply natural phrases and their meanings immediately in algorithm definition. In order to make inferences from a natural language text, we do not need a *prior* semantic knowledge, expressed mathematically. We only need to supply our algorithm with the patterns of correct natural language reasoning.

We can use natural language words and concepts in a computer program almost as fluently as in human speech. We only need to make patterns of the phrases, where these words and concepts are correctly demonstrated, and include them into our program.

We can apply patterns of unrestricted texts immediately in reasoning, consistently with the meaning of these patterns. According to the method of implicit semantics, we can define and manipulate with this meaning implicitly, without reference to any mathematical construction known a priori.

So, we need to supply our algorithms with the generalized and structured examples of correct reasoning. We relate to a future research the methods of automatic extraction and composition of such example patterns into the theories for performing specific tasks. This is discussed in Sections 6 and 7.

The principal limits of an *artificial* formalization, such as that depicted in the path 3-4-5-6-7-8 of Figure 2, lie in the need to invent and accommodate the heterogeneous and complex artificial formalisms. Indeed, the more diverse and complex constructions we invent, the harder are these constructions to combine. Thesis 3.1 and the method of implicit definition of senses provide some hope that these limits can be overcome by replacing the artificial mechanisms with the clauses composed from the generalized examples of *natural* texts, where the senses are already combined in a natural way.

An interesting implication of this is the possibility to implicitly define and apply the meaning of idiomatic expressions. We consider this a perspective research direction for natural language processing. Other interesting directions include: 1) Generating a large question answering system from question and answer examples and a background knowledge expressed in a text form; 2) Creating an encyclopedia of *working* algorithms, which are described in common-sense verbal terms, as we do in the following section.

5 Case Study: FOPC Specifics as a FOSC Theory

The following string theory example (Example 5.1) demonstrates an algorithm for logic formula transformation where some new object variables of a *predicate* calculus are automatically generated. Consider the following well-known rule for transformation of formulas in a first-order predicate calculus with quantifiers:

Formula (Qx)F ∨ (Qx)H of the calculus can be transformed into formula (Qx)(Qy) (F ∨ G), where:

Q is a quantifier ∀ or ∃;
Formulas F and H do not contain variable y;
Formula G is the result of global replacing of variable x in formula H with variable y.

Example: formula (∀x)a(x,y) ∨ (∀x)b(x,t) can be transformed into formula (∀x)(∀z) (a(x,y) ∨ b(z,t)).

This is essentially an *informal* description, which contains some *formal* elements in a textual form. We do not intend to study this predicate calculus: we use the above description only for demonstration of our methods of extracting algorithms *immediately* from string examples, which may contain formal and informal fragments.

This description defines a transformation of logic formulas. The following string theory does this transformation using only the generalized string examples of this transformation as a building material. A working Sampletalk version of this theory can be found in our website (see below) along with an experimental implementation of Sampletalk compiler. Note the prefix "¬" in the 4th clause, which denotes a negation as failure.

Example 5.1. A string theory for logic formula transformation.

% Goal. Variable W stands for the result of the transformation:

the result of shifting quantifiers in formula (∀ x 0) [a (x 0, y)] ∨ (∀ x 0) [b (x 0, t)] is formula W .

% The main theory clause:

the result of shifting quantifiers in formula (Q X) [F] ∨ (Q X) [H] is formula (Q X) (Q Z) ([F] ∨ [G]) :–

X is notation for object variables ,,
Z is notation for object variables ,,
formula F does not contain object variable Z ,,
formula [G] is the result of replacing X by Z in formula [H] ..

% Definition of the variables for the predicate calculus:
x 0 is notation for object variables ..
X 1 0 is notation for object variables :–
X 0 is notation for object variables ..

% A straightforward explanation what is "does not contain" and "contains":
formula F does not contain object variable Z :–
¬word F contains word Z ..
word A X B contains word X .. *% Note free FOSC variables A and B here*

% Algorithm for replacing variables in the predicate-logic formulas:

formula [A Y N] is the result of replacing X by Y in formula [A X M] :–
formula [N] is the result of replacing X by Y in formula [M] ..
formula [A] is the result of replacing X by Y in formula [A] :–
formula A does not contain object variable X ..

This theory produces the following output: the result of shifting quantifiers in formula (∀ x 0) [a (x 0, y)] ∨ (∀ x 0) [b (x 0, t)] is formula (∀ x 0) (∀ x 1 0) ([a (x 0 , y)] ∨ [b (x 1 0, t)]). Remind that square brackets [and] are special symbols (not terminal symbols), which define a nesting structure on strings and constrain the possibilities of string unification. Sometimes there are too many possibilities of unifying texts with variables. The nesting structure constrains this.

Constructing such a theory, we do not have to think of the arrays, formal grammars, parsing, loops, logical conditions and procedures, necessary in a traditional programming. We do not have to think of logical relations, too, although we build string clauses with a logical behavior. We only think of and deal with string examples. The above theory can be similarly developed into a more sophisticated logic formula transformation algorithm.

In a way, this example shows a hybrid approach, where we *implicitly* define and apply a strict formal calculus syntax via the generalized examples that only *demonstrate* this calculus. In a similar and seamless way we can embed any system of formal notations into the example-based reasoning, according to Thesis 3.1. So we can combine the notations of any formal theory or ontology, which is somewhere defined *explicitly*, with natural language reasoning.

6 Experiments with Sampletalk Compiler

Sampletalk language [6] is an implementation of FOSC. An experimental Sampletalk compiler, along with the working examples described here, currently can be downloaded

from website www.sampletalk.com. In this compiler we apply a convention that the string segments, used for the substitution of variables in text unification, cannot be empty. The compiler is implemented using SWI-Prolog and includes some Prolog specifics: machine-oriented constants, a cut operator etc. Currently only this experimental implementation of FOSC exists. About 30 working examples of Sampletalk programs can be found in our works (see [6] and the references therein). A large Sampletalk program for an experimental linguistic application can be found in [24].

Sampletalk (FOSC) Programming vs. Prolog (FOPC) Programming. Our examples show several essential differences between Sampletalk programming, based on the alignment of generalized strings, and Prolog programming, based on the artificial predicate notations.

First of all, we do not need the embedded predicates *append/3, member/2, arg/2, var/1, atom/1, univ/2* etc, applied in Prolog for the analysis and transformation of the data between different representations (lists, predicates, clauses, strings etc). These predicates, just like the basic operators of any conventional programming language, form the basis of the *artificial* formalization in Prolog programming. In Sampletalk the roles of the elementary operators play these more expressive and intuitive *natural* expressions*: is, in, by, the result of, does not, for* etc. What is important for us, the interaction of program clauses in Sampletalk can be governed by such expressions consistently with their natural common-sense meaning. Furthermore, we can apply as many natural expressions as we want. This is impossible in Prolog.

The second difference is that, given the informal problem description and the examples of desired processing, *we do not invent or write the program code*. We *embed* the examples and the description fragments immediately into the program, trying to preserve their text form as much as possible. (Actually, we do only string generalization, structurization, and some adaptation of the texts in order to comply with FOSC notation). This is also impossible in Prolog, unless the problem in question is already described in a predicate form.

As the other side of the coin, in Sampletalk we need a strict matching of the string patterns, which a priori are unrestricted. This may impose a problem if the patterns contain commas, capitalizations and other variations, which often are unnoticed when a human deals with a natural language text. Prolog compilers do not allow such syntax freedom and prevent the programmer from inserting the inconsistent symbols into a program code. In most cases this problem can be easily overcome by a preprocessing of the text in question, as we did in our examples. Some other differences between Sampletalk and Prolog programming are discussed in Section 4 and in paper [6].

7 Discussion

This work contains only an initial investigation of the possibility to analyze unrestricted texts via matching of the text fragments. Some questions, raised by this work, can be answered only regarding specific applications, e.g., natural language processing systems. Here we do not describe any complete system of this kind and do not compare our approach to others in terms of performance, coverage and disambiguation. Among the questions, we should ask the following: How to choose text fragments to be considered as the strings? How to automate the generalization and

structurization? How to choose the string pairs to build a theory from? How to generate a FOCS theory with a desirable behavior? Can we apply the existing example-based machine learning methods in order to build a FOSC theory?

Here we only outline some useful directions. More substantial answers to these questions are related to a future research.

Extracting a String Theory from String Examples. FOSC theories can be extracted from string examples using some machine learning methods. We already know that the generalized string patterns can serve as a building material for constructing such theories. Using simple statistical methods we can prepare a large set of generalized string patterns from a raw corpus of natural texts. Then we can try to apply this material for the automatic construction of FOSC theories for specific applications. In this framework, the problem of generating a FOSC theory P can be formulated as follows.

Consider the following two sets of string pairs, which represent the desired and undesired input-output pairs respectively:

$E_+ = \{(s_i, t_i): i=1,2,\ldots, m\}$ *(Positive examples)*
$E_- = \{(s'_j, t'_j): j=1,2,\ldots, n\}$ *(Negative examples)*

A notation $Q(s)=t$ will be applied below for denoting a theory Q, which produces an output string t from an input string s in a limited time. A theory P can be automatically generated using the following objective function:

$$f(Q) = (|\{i: Q(s_i) = t_i\}| - |\{j: Q(s'_j) = t'_j\}|) / |Q|$$
$$P = \arg\max(f(Q))$$

where $|Q|$ denotes the total number of strings contained in the clauses of theory Q. The numerator of the objective function characterizes a quality of the theory Q regarding sets E_+ and E_-. The denominator characterizes a complexity of the theory Q. This objective function can specify how to choose among the many ways to structure and generalize a given set of strings and how to join the strings into clauses. Some modern optimization methods can be applied for the maximization of $f(Q)$.

We can use various sources of string samples for inclusion into a theory Q. For example, we can create such samples using a pair-wise alignment and most specific generalization of sentences, contained in a natural language corpus. Consider the following set of sentences (here we avoid capital letters so that they could be used for FOSC variables):

a book is in an open box
a notebook is in the closed box
the cat is in the cage
a fox is in a zoo
john is in his red car
this large table is red

A pair-wise alignment of these sentences provides a set of generalizations, which contains the following strings:

a [X] is in [D] [Y] box,
[X] is in [D] [Z],
a [X] is in [Z],
[X] is in [Z],
[X] is [A],

where variables X and Z stand for a noun or a noun group, D stands for determiners, Y stands for adjectives, and A stands for adjective phrases. The generalization is done by a substitution of the non-matching text constituents with variable symbols. Note that we get so generalized sentence patterns without any grammar or morphology models.

The discussable subject, suggested by these examples, is a possibility to construct a practical question-answering system using such material extracted from a large raw corpus of sentences. This section should be considered an invitation to carry out experiments in order to analyze this possibility for various problem domains, e.g., for developing a library-related dialogue system about book availability, or a system for a dialogue about geographical objects.

An important observation, supported by our examples, is that we don't need the explicitly defined qualifiers (formal grammar non-terminals): a noun-phrase, an adjective phrase etc. for the variables X, Z, A etc, although the variables may match only phrases of such types when the theories are applied to correct inputs.

For example, if a simple and correct English sentence is aligned with string "[X] is in [Z]", then we can expect that variables X and Z represent noun phrases, *although we do not explicitly define what a noun phrase is.* As a curious implication of this observation, we have a certain freedom from the ambiguity problems related to the usage of formal grammars. In fact, we don't apply formal grammars at all: FOSC provides the alternative options for text analysis and generation.

However, the multiple possibilities of alignment and unification of strings can impose their own ambiguity problems. For example, when string "[X] is in [Z]" is aligned with a complex raw sentence, the variables X and Z may represent not only noun phrases, but also some non-classifiable sentence fragments. We can approach to these problems using FOSC methods, without the application of external mechanisms for resolving ambiguity. For example, when aligning an English sentence with string "[X] is in [Z]", we can require that the text constituents for variables X and Z do not contain still non-structured substrings, which contain segment "is in".

Below we outline another direction for building FOSC theories. Here the composition of strings into clauses is controlled by a set E of ground string examples and a subset $G \subset G(E)$ of its generalization universe (see Definitions 2.5 and 2.10). Let us apply a notation $H \in\in G$ for stating that string H is a representative of some element of set G. Remind (Section 3) that a string theory is considered a *sequence* of clauses.

Algorithm Scheme 7.1. Build a string theory *Q* of a set E of ground string examples using generalizations $G \subset G(E)$.

1) Take the most general element $H \in\in G$, not considered before, and put the clause H (with empty body) at the top of sequence Q. Stop stating an unsuccessful status if such element H does not exist.
2) If Q is a theory of set E, then remove all its clauses, which are not used in the application of Q to the elements of set E, and stop stating a successful status.
3) If Q contains a clause with empty body, find a bounding sequence B for its head H using a supporting example $\alpha \in E$. Proceed to Step 1 if this is impossible. Otherwise replace this clause with the description clause H :– B. Put into Q all the attractors of this bounding, which are still not the heads of clauses in Q, in the form of clauses with empty bodies. Proceed to Step 2.

We justify our terminology of acceptors and attractors as follows. The heads of the clauses being created should *accept* the examples. The heads of already created clauses should *attract* the bounds when constructing the bounding sequences. In Step 3 we assume that the attractors and the bounding sequences are constructed from the representatives of elements of set G.

In order to apply this scheme, we should specify the following: 1) how a bounding sequence B is constructed; 2) in what positions of the theory P the new attractors are inserted; 3) how to determine whether a current theory is a theory of set E. The formation of set $G \subset G(E)$ should also be specified.

A possible strategy for the search of description clauses consists in the requirement that any description clause H :– B is supported by a *majority* of examples $\alpha \in E$, for which this clause is applicable. When sets E and G are sufficiently small, we can apply an exhaustive search among all possible bounding sequences and positions for inserting the attractors.

Example 7.1. Consider the ground string examples and four upper elements of the generalization universe described in Fig. 1 (Section 2). Assume that we have only three variables: A, B, C, and that we look only for clauses containing no more than one bound. In this case Algorithm Scheme 7.1 transforms into a simple exhaustive algorithm. The string theory of Example 2.1 is found by trying all possible bounding clauses and positions of the attractors.

In more complex cases we should seek for a partition of set E into simpler parts or for the application of special heuristics. In the tradition of ILP [18], the search for useful hypotheses can be combined with the application of background knowledge.

The search strategy of Algorithm Scheme 7.1 can be characterized as follows:

Try to apply the most general text pattern as a clause. If the pattern is too general, try to bind it into a description clause for some of the examples. If the theory still is not constructed, then put less general patterns on the top of the theory so that those too general become unattainable for the specific cases.

Generalization universe (cf. Section 2), derived from sentences of a natural language corpus, defines an ultimate expressive structure of the knowledge, represented in the corpus, without reference to any artificially formalized concepts. Doing a pair-wise alignment and generalization of the sentences, contained in the corpus, we can expect that the frequency of the shorter generalizations is higher and that the shorter generalizations represent more correct, more general, and more applicable sentence patterns, like those applied in our examples in Section 4.

It is unrealistic and probably unnecessary to try to build the entire generalization universe G(E). For the natural language sentences, the set $G \subset G(E)$ can be limited in such a way that only the short and simple sentences are applied for the pair-wise generalization.

So, FOSC provides a universal framework of text perception, formalization, understanding and manipulation, which is independent of any prior formal knowledge. This framework allows us to immediately exploit the regularities and meanings, *implicitly* demonstrated in the text.

Here we see a philosophical challenge to the common usage of *explicit* methods, which often appear to be stiff, expensive, incomprehensible and mutually incompatible. Paradoxically, today the *implicit* methods are almost unknown, and most researchers define the semantics only explicitly, involving a large number of complex and hardly compatible formalisms resulted from the available models of various phenomena. In contrast to this, we try to develop a *universal* framework of intelligent text processing, which is independent of any prior knowledge.

There is another philosophical challenge, which is related to the usage of logic reasoning in general. Can we give up the logic connectives in the same way we have given up the logic predicates? In FOSC, we apply the traditional logic connectives for constructing and combining clauses. More specifically, we apply conjunction, negation, implication, and the resolution rule. So we manipulate with text semantics without predicate and formal grammar mechanisms. What will we gain if we remove the logical connectives as well? Can we model the text meaning using *only* the generalization, structurization and unification of text examples? Can we organize a *natural* reasoning, appealing only to the inherent logic of the text itself, without the first order logic inference? We finish this paper with this question unanswered, addressing it to a future research.

8 Related Work and the Ideas for Future Research

Modern information resources, such as Wikipedia, digital libraries, patent banks and technical regulations, contain a large amount of knowledge in the form of unstructured or semi-structured texts. In order to make inferences from this knowledge, attempts are made to structure it into a first order logic form. A common usage of logical methods assumes the design of logical predicates. This is done, for instance, in CycL [14] system. CycL contains a very large set of handcrafted first order logic micro-theories. However, this is a very expensive undertaking. Besides, the predicates reflect the view of their designer rather than the inherent logic of the subject.

We show that one can do first order logic without predicates at all, providing, in this way, a new direction for tackling this problem. The *implicit* definitions are much simpler to combine than the *explicit* definitions. Indeed, as in our speech, we can easily combine thousands of concepts without defining them explicitly. Furthermore, the way we apply the concepts assumes that they are already combined in some existing and correct text examples. In contrast, the attempts to combine *explicit* concepts typically require an expensive manual encoding and very complex axiomatization.

String unification enables us to implicitly operate with senses, related to the aligned text constituents. Unification of various kinds of structures (which have slots or variables for filling-in by the matching constituents) is applied in various fields. Unification of logical atoms is the most prominent example. Unification has been applied not only in a Logic Programming framework, as we do, and not only for strings or logical atoms.

Unification grammars (see, e.g., [10], [11]) present an example where a unification of special objects (feature structures) is applied in order to constrain the application of formal grammar rules. Note, however, that in this research the unifiable structures play a secondary role in reasoning. The primary role has the formal grammar. In a

Logic Programming framework the unification of objects has a more active role, supporting the implicit definition and manipulation of a larger class of senses.

Implicit formalization methods have their own history. In mathematical equations the implicit variables may denote functions, which do not have any explicit definition, or for which the explicit definitions are very complex and should be avoided. The following programming languages support the implicit definition of various senses by instantiation of variables through a matching of various structures. ANALYTIC language [7] contains special operators *compare* and *apply*, which are applied for the matching of symbolic structures containing variable symbols. For instance, the application of a symbolic form of differentiation rules to an analytic expression of a function produces an analytic expression of the function derivative. Matching of various structures, containing variables, is applied in Snobol [8], Refal [23] and Planner [9] languages. FOSC can be considered an attempt to extend these approaches for working with unrestricted strings.

ILP [18] suggests substantially developed inductive methods for the inference of predicate-logic programs from examples of logical atoms and a background knowledge, which is also expressed in a predicate-logic form. Grammar induction using ILP methods (LLL, Learning Language in Logic) is described in [5]; see also [4]. ILP methods can be similarly applied for the automatic extraction of FOSC theories from string examples. We consider this a prospective direction for future research. Some consideration of ILP-style methods for text-related inference is done in [6]. There is a certain analogy between a most specific common generalization of strings (Section 2) and a least general generalization of logical atoms, introduced by Plotkin [21].

An extensive analysis of various unification algorithms, applied in logical systems (including string unification algorithms), is given in [1]. In [13] an extension of classical first order logic with sequence variables and sequence functions is described. These constructions coexist with ordinary variables and functions of a fixed arity. A special syntax, semantics, inference system and inductive theory with sequence variables are developed in this calculus. G.S.Makanin [16] proved the computability of a word equation (string unification) problem in 1977. NP-hardness of this problem is shown in many publications, see e.g. [2]. We apply a limited, but efficient string unification algorithm, which is sufficient for achieving Turing completeness. Various methods of string alignment are developed and applied in biotechnology and NLP.

The incorporation of common-sense reasoning into computer programs is currently an active research area. For instance, an attempt of this kind, related to building a large ontology, is done in SUMO [19] project. Our method of implicit definition of senses is another attempt to apply existing common-sense reasoning patterns in computer programs.

An *automatic* extraction of FOSC theories from texts, contained in the information resources mentioned above, is a perspective direction for the future development of the method. Thesis 3.1 suggests that, given a sufficiently rich set E of string examples which *demonstrate* a theory or algorithm, we can always build this theory or algorithm from the generalizations of the examples. The generalization universe G(E) (cf. Section 2) provides all possible building blocks for such a theory.

So, the direction for a future research, which we suggest here, is finding a way of extracting and composing such blocks into a theory automatically, without inventing *artificial* notations for the designation of the involved concepts and relations.

This direction will hopefully provide new insights for the analysis of biological sequences: since we apply the unrestricted string patterns immediately in reasoning, we may compose a FOSC theory, which employs the inherent meaning of such sequences and does not depend on any previous formal knowledge.

9 Conclusion

First Order String Calculus (FOSC) is an attempt to build a cognitive model of the brain by simulating the matching, generalization, structurization and composition of the images contained in our memory. FOSC can be applied for the extraction of formal theories and algorithms immediately from unrestricted string examples, without any previous semantic knowledge, expressed mathematically. The regularities, demonstrated in similar text examples, are extracted using an alignment and a generalization of the examples. The algorithmic forms of these regularities – string generalizations, structurizations and compositions into clauses – simultaneously serve as elements of the theory of the examples and as a universal algorithm definition means. This way of creating algorithms is Turing-complete and provides a reasoning framework, which is at least as powerful as a predicate logic, based on the axiomatic principles. Furthermore, FOPC is a special case of FOSC.

FOSC theories provide an alternative to both predicate-logic-based and formal-grammar-based approaches to text analysis and generation.

The method of implicit semantics, based on string alignment, is an adaptation of the traditional method of implicit function definition in mathematical equations, for the fields of logics, linguistics, and intelligent text processing in general. Like in traditional mathematics, this method allows one to apply the inherent meaning of the expression constituents and avoid the complex formulas, which are necessary when the meaning is defined explicitly.

FOSC, along with the generalization universe of a set of unrestricted strings, can be considered a kind of concept algebra, where the concepts are defined implicitly. The generalization universe can be applied in order to assess the expressive power and cognitive limits of the knowledge, expressed in the unrestricted texts, independently of any other knowledge.

References

[1] Baader, F., Snyder, W.: Unification Theory. Handbook of Automated Deduction, ch. 8. Springer, Berlin (2001)

[2] Černá1, I., Klíma, O., Srba1, J.: On the Pattern Equations. FI MU Report Series, FIMU-RS-99-01 (1999)

[3] Church, A.: Introduction to Mathematical Logic, vol. 1. Princeton University Press, Princeton (1956)

[4] Cicchello, O., Kremer, S.C.: Inducing Grammars from Sparse Data Sets: A survey of Algorithms and Results. Journal of Machine Learning Research 4, 603–632 (2003)

[5] Dzeroski, S., Cussens, J., Manandhar, S.: An Introduction to Inductive Logic Programming and Learning Language in Logic. In: Cussens, J., Džeroski, S. (eds.) LLL 1999. LNCS, vol. 1925, pp. 4–35. Springer, Heidelberg (2000)

[6] Gleibman, A.H.: Knowledge Representation via Verbal Description Generalization: Alternative Programming in Sampletalk Language. In: Workshop on Inference for Textual Question Answering, July 2009, 2005 – Pittsburgh, Pennsylvania, pp. 59–68, AAAI 2005 - the Twentieth National Conference on Artificial Intelligence, http://www.hlt.utdallas.edu/workshop2005/papers/WS505GleibmanA.pdf
[7] Glushkov, V.M., Grinchenko, T.A., Dorodnitcina, A.A.: Algorithmic Language ANALYTIC-74. Kiev. Inst. of Cybernetics of the Ukraine Academy of Sciences (1977) (in Russian)
[8] Griswold, R.E.: The Macro Implementation of SNOBOL4. W.H. Freeman and Company, San Francisco (1972)
[9] Hewitt, C.E.: Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in a robot. Technical Report, AI-TR-258, MIT Artificial Intelligence Laboratory (1972)
[10] Jaeger, E., Francez, N., Wintner, S.: Unification Grammars and Off-Line Parsability. Journal of Logic, Language and Information 14, 234–299 (2005)
[11] Jurafsky, D., Martin, J.H.: Speech and Language Processing. An introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. Prentice-Hall, Englewood Cliffs (2000)
[12] Kleene, S.C.: Mathematical Logic. John Wiley & Sons, Chichester (1967)
[13] Kutsia, T., Buchberger, B.: Predicate Logic with Sequence Variables and Sequence Function Symbols. In: Asperti, A., Bancerek, G., Trybulec, A. (eds.) MKM 2004. LNCS, vol. 3119, pp. 205–219. Springer, Heidelberg (2004)
[14] Lenat, D.B.: From 2001 to 2001: Common Sense and the Mind of HAL. In: Stork, D.G. (ed.) HAL's Legacy: 2001's Computer as Dream and Reality. MIT Press, Cambridge (2002)
[15] Lloyd, J.W.: Foundations of logic programming. Artificial Intelligence Series. Springer, New York (1987)
[16] Makanin, G.S.: The Problem of Solvability of Equations in a Free Semigroup. Mat. Sbornik. 103(2), 147–236 (in Russian); English translation in: Math. USSR Sbornik 32, 129–198 (1977)
[17] Markov, A.A.: Theory of Algorithms. Trudy Mathematicheskogo Instituta Imeni V. A. Steklova 42 (1954) (in Russian)
[18] Muggleton, S.H., De Raedt, L.: Inductive Logic Programming: Theory and Methods. Logic Programming 19(20), 629–679 (1994)
[19] Niles, I., Pease, A.: Towards a Standard Upper Ontology. In: Welty, C., Smith, B. (eds.) Proceedings of the 2nd International Conference on Formal Ontology in Information Systems (FOIS 2001), Ogunquit, Maine (2001)
[20] Piaget, J., Inhelder, B., Weaver, H.: The Psychology of the Child. Basic Books (1969)
[21] Plotkin, G.: A note on inductive generalization. Machine Intelligence, vol. 5, pp. 153–163. Edinburgh University Press (1970)
[22] Robinson, J.A.: A Machine-oriented Logic Based on the Resolution Principle. J. ACM 12(1), 23–41 (1965)
[23] Turchin, V.F.: Basic Refal. Language Description and Basic Programming Methods (Methodic Recommendations), Moscow, CNIIPIASS (1974) (in Russian)
[24] Vigandt, I.: Natural Language Processing by Examples. M. Sci. Thesis, Comp. Sci. Dept., Technion, Haifa, 115 p. (1997) (in Hebrew, with abstract in English)

Appendix: The Applied String Unification Algorithm

In this appendix we describe a string unification algorithm, which is applied in our implementation of Sampletalk compiler [6]. Let (S_1, S_2) be a pair of strings, which can be represented by any of the following alignments (here the indexes define the correspondence between the aligned segments):

$$\begin{aligned} S_1 &= s_1 V s_3 \\ S_2 &= s_1 g_2 g_3 \end{aligned} \tag{1}$$

$$\begin{aligned} S_1 &= s_1 s_2 s_3 \\ S_2 &= s_1 W g_3 \end{aligned} \tag{2}$$

Here *V* and *W* are variables, which do not occur in segments g_2 and s_2 respectively, and segments s_3 and g_3 are either empty or start with compatible symbols, which means that either $s_3(0) = g_3(0)$ or at least one of these symbols is a variable. (We apply a 0-based indexation of the symbols in strings). Segment s_1 also may be empty. Segment g_2 (s_2) is assumed to be a valid string (concerning the nesting structure, defined by the square bracket pairs) and also may start from a variable symbol. Variables *V* and *W* are called *bound* by segments g_2 and s_2 respectively.

Such an alignment is called *a leftmost bounding hypothesis* for string pair (S_1, S_2). There can be more than one leftmost bounding hypothesis for the same string pair. As an example, consider two alignments of string "[A , B] ⇒ [C , A]" with string "[c a t , d o g , m o n k e y] ⇒ [W]", described in Section 2, where variable A is bound by segment "c a t" in one case and by segment "c a t , d o g" in another.

Given a string pair (S_1, S_2), we consider all leftmost bounding hypotheses sequentially according to the following convention:

a) If $g_2(0)$ is a terminal symbol (case 1), then the versions for g_2 are ordered by incrementing the number of symbols in segment g_2;
b) If $s_2(0)$ is a terminal symbol (case 2), then the versions for s_2 are ordered by incrementing the number of symbols in segment s_2;
c) Otherwise (a combined case where both $g_2(0)$ and $s_2(0)$ are variables) we consider all versions for g_2, organized similarly to the case (a), followed by all versions for s_2, organized similarly to the case (b).

Since the number of symbols in a string is finite, the sequence of leftmost bounding hypotheses is finite for any string pair (S_1, S_2). The following recursive procedure sequentially produces the unifications of string pair (S_1, S_2):

Procedure 1. (String Unification Algorithm): Find unifications of a string pair (S_1, S_2).

1) *If* $S_1 = S_2$ *then report string* S_1 *and exit.*
2) *Find a leftmost bounding hypothesis, not considered before, for the string pair* (S_1, S_2). *If such hypothesis does not exist then exit.*
3) *Replace all occurrences of the bound variable in string pair* (S_1, S_2) *with the corresponding bounding segment.*
4) *Apply Procedure 1 to the string pair* (S_1, S_2)*, modified in Step 3.*
5) *Undo the change made in Step 3 and proceed to Step 2.*

We assume that, along with reporting every unification string, this procedure reports the corresponding substitution of variables, which is applied for the original string pair (S_1, S_2) in order to produce this unification string.

We should note that the presented algorithm has certain limitations. Particularly, we cannot be sure that the algorithm produces *all* unifications of a given pair of string. In spite of these limitations, the algorithm can be applied in a Turing-complete algorithm definition system (see Theorem 3.1).

String unification algorithm, defined above, organizes all the possibilities to unify a pair of strings in a specific order, which we call *a canonical order*. The unifications are ordered and can be sequentially applied according to this order. All possible instantiations of a clause H :– B by a literal S (if there are many) are also ordered according to the canonical order of possible unifications of strings S and H. This order is used in the application of a string theory to a string (Algorithm 3.1).