ISSN 2007-9737

Logic Abstraction Operations and their Algorithmic Implementation

Pavel Zheltov

Kazan Federal University,
Research Laboratory of Text Analytics,
Kazan

PVZheltov@kpfu.ru

Abstract. The article deals with logic abstraction operations, such as isolation, identification and generalization and their algorithmic implementation using the meta-programming language *Sympl,* which is being developed by the author. As part of the implemented logic operations, new data types such as “identification set", "concept", “notion” and "category" were implemented. The data type “identification set” represents sets, the elements of which all have either common properties or relations and are the result of the application of identification operation to logical objects. The data type “concept” is used for representation of concepts that are results of application of identification and generalization operations and is represented by two daughter data types (subtypes): “notion” and “category”. The “notion” data type represents the result of application of abstraction of generalization to an identification set. The application of abstraction of generalization two (or more times) results in a “category” data type - an extremely broad notion. The developed algorithms can be applied in text analysis when words are presented as logical objects: for finding synonyms, functionally similar personages or objects by their description and activities and so on.

Keywords. Logic programming, language analysis, isolation, identification, generalization, set, notion, category.

1 Introduction

Whereas last trends in logic programming’s development were towards high-order logic, this theory and its implementation contributing of course to logic programming, some other ideas were left aside. Such was the theory of abstractions derived from classical logic, that was under focus in the theory and practice of logic and functional programming in late 80-s and early 90-s of the XX century, see, for example [4, 5], but was abandoned soon after and did not receive any further development and implementation.

This theory, however, must not be neglected as its implementation could contribute to logic programming and AI research as a whole. The fact is that abstraction is a very important cognitive function and is proper to human and all highly developed organisms with creative activity, such as most of mammals and even birds.

Strangely enough the words “abstraction of isolation” or “isolating abstraction”, “abstraction of identification” or “identifying abstraction”, “abstraction of generalization” or “generalizing abstraction”, “concept”, “notion” and “category” are ignored in many dictionaries of logical terms, including the Oxford’s Dictionary of Logic and most of authors deal with it without any further classification of its operations, see for example [1]. For an exposé of a general mathematical theory of set abstracts see [3].

In this paper we consider main types of abstraction proper to cognition based on their application to language, as the results of human mental activity are (or can be) expressed through words (and language), and have tried to implement the logical process in terms of mathematical definitions and algorithmization.

It was shown that abstractions of isolation and identification can be well formalized for any area of their application (not only language) in terms of classical logic and algorithmized as well, while the formalization of the abstraction of generalization encounters some difficulties, as in order to formalize it one must use special procedures to

identify properties or relations that have much in common, but are not completely equal, as well as either create new words for new notions and concepts resulting from the application of this operation or use an ordinal denotation system, based on letters and numbers.

One may ask however two questions on the account of the new theory proposed in this paper – why is it necessary to develop a new data type theory and why it isn't possible to implement it in terms of existing programming language?

The answer to these questions will be the following – neither Prolog, nor Lisp, as well as other declarative and imperative programming languages, that were designed for logic and symbol programming, as well as standard programming languages, such as C++, Pascal and others, do support operations that are necessary to implement this theory, such as random selection of a set's element and dynamical declaration of a function with a dynamically defined name, that is needed when returning a property or a relation of a logic element as a function.

In fact, even C++ - a programming language that allows to handle variables and data in most virtuous way doesn't allow to convert dynamically an array element of a string/char type to a function with the same name as its value.

The syntax of a new meta-programming language *Sympl* that is being developed by the author for this purpose allow to do all these things< i.e. it supports type conversion for variables of "logic" and "predicate" types that are being introduced, based on their logical and functional compatibility and not on a calculative one as in C++ or Pascal.

2 Mathematical Definitions

In the process of cognition, human intelligence identifies individual properties, possessed by the objects of cognition and the relations between these objects, and begins to operate them as if they existed independently of these objects. Human intelligence is also peculiar to group cognition objects on the basis of the allocation of their identical properties or relationships.

In both cases, we are talking about abstractions, while in the first case – about the *isolating abstraction* (otherwise called abstraction of isolation), and in the second case – about the *identifying abstraction* (otherwise called abstraction of identification) and the results of these abstractions are called in logic *abstract objects* [2].

Since in the process of thinking (mental activity) human intelligence, along with images, operates with words, and the results of mental activity are also expressed through words, the results of abstractions are reflected in the lexicon of the natural language.

Thus, as examples of the result of isolating abstraction can be considered such words as '*whiteness*' (property), '*kindness*' (property), '*friendship*' (relation), etc.

And as examples of the result of the abstraction of identification can be considered such words as '*good people*' (on the basis of the property '*to be good*'), '*friends*' (on the basis of the relation '*to be in friendly relations/friendship with someone*'), etc., which denote the corresponding set of cognitive objects. These sets are called identification sets.

In logic, the abstraction of identification, according to which each predicate can be matched with a certain set and vice versa, is implemented by the so called "collapse axiom".

Symbolically, the collapse axiom is written as $((x \epsilon M) \equiv P(x))$.

This entry means that, based on the fact that some objects (x) share a common property (P), they can be grouped into a separate set (M). The converse is also true: a set (M) that consists of some elements (x) corresponds to/can be matched with a certain property (P) that is shared by the set's elements (x).

If one designates the operation of isolating abstraction as A_{ISOL}, then its application to some object x can be written as:

$$A_{ISOL}(x) = \begin{cases} P_i(x), & i = \overline{1,n}; \\ Q_j(x, y_j), & j = \overline{1,m} \\ \varnothing, \end{cases} \quad (1)$$

where $P_j(x)$ is a property from the set of properties of the object x, $Q_j(x, y_j)$ is a relation from the set of relations of the object x, and y_j - an object that is in a relation (Q_j) with x. The application of the isolating abstraction A_I to some object x, in case when it has no defined properties or relations, will

result to an empty set, which is denoted in (1) by the symbol 'Ø'.

If we denote the operation of identifying abstraction as A_{IDENT}, then its application to some sequence of objects (x_1,..., x_n) can be written as:

$$A_{IDENT}(x_1, \dots x_n) = M_i : \forall M_i \equiv Pi(x) \cup Qi(x, y), x \in M_i i = \overline{1, m}. \quad (2)$$

3 Algorithmic Apparatus

In this work a meta-programming language called *Sympl*, that is somewhat similar to *Pascal*, is used for algorithmic implementation. The scope of the paper doesn't allow to present its syntax in details, that is why syntagmas that are alike those of the standard *Pascal* are not commented. Only those that are peculiar to *Sympl* are explained. The meta-programming language *Sympl* is a context-free language.

Before proceeding to the implementation of these logical operations in the framework of the *Sympl* programming language, let us define the type "*logic*" (a logical type) as:

```
(declare) type "logic" as record of
P: array of string;
Q: array of record self: string,
where Q [i: integer] << self
y: symbol;
end;
end.
```

Or as:

```
(declare) type "logic" = record of
P: array: string;
Q: array: record of self: string,
where Q[i] << self,
where i: integer;
y: symbol;
end;
end.
```

In the declared syntagmas, the logical type is specified as a record containing an array of properties (*P*) and an array of relations (*Q*).

The reserved word "*declare*" before the declaration of a type in *Sympl* is optional. There are following syntagma types in *Sympl*: declarative ones (that declare types or variables), directive ones (that give to the compiler of the symbolic environment directives on how to declare or execute something), executive ones (straight commands or operations), interrogative ones (questions to the symbolic environment) and affirmative ones (logical affirmations).

The directive, executive and affirmative syntagmas form the so-called "imperative" part of the *Sympl*'s syntax.

For example, the two above syntagmas are declarative ones, but have a directive expression with the construction "*where* …" ("*where Q[i: integer]<<self*" and "*where Q[i]<<self*, *where i: integer*" respectively) in each of them, that tells the compiler to replace (using the right to left replacement operation expressed by the "<<" symbol) the "*Q[i]"* expression (actually the address of the records' array's index expressed by *Q[i]*) with the variable "*self"* of this records array's index (actually with the string stored in it). Thus, the variable *self* becomes a sort of alias of the *Q[i]*.

In most of existing programming languages this is impossible. For example, in standard *Pascal* (and in the *Object Pascal* as well) one has to declare the "logic" type this way:

```
type logic = record
P: array of string;
Q: array of record
relation_name: string;
y: ^logic;
end;
end;
```

Thus, one has to introduce the variable *relation_name* to store the names of the relations and refer if needed to the name of the *i*-th relation stored in an array of relations Q of a variable of a logic type, by the following, far not most convenient way:

Q[*i*].*relation_name*;

As a result, the system will give out the value stored there, for example the string '*friendship*'.

Since in this work, due to its limited volume, it is not possible to cite all the production rules of the *Sympl*'s syntax (that is yet not complete) we comment on each of the presented syntagma and point out its peculiarities if needed.

In order not to return, however, each time to the explanation of some basic features of *Sympl*'s syntax in declaring names, we will explain a number of nuances in naming conventions.

A name in *Sympl* (as in Pascal) is a sequence of letters and numbers (from '0' to '9') beginning

with a letter. The '_' sign can occur inside a name, but should not occur in its beginning or end, because this sign is used as a substitute for the names of functions, procedures, variables, and types (fulfills the same function as a template in C++).

Also, a complete syntagma (function, procedure, unit or program) in *Sympl* must end with a '.', rather than a semicolon (as in most of programming languages) which corresponds to a general declarative approach strategy, where each complete syntagma can in principle be an independent piece of code ready for execution.

When one retrieves any property or relation from logical objects and creates independent predicates (implemented as functions) with names that match the names of these properties or relations (that are stored as strings in the P and Q arrays), the dereferencing operation '<…>' is used: <P[i]> (<parameter list>) or <Q[i]> (<parameter list>).

4 Algorithmic Implementation

The operation of isolating abstraction for logical objects (of the logic type), thus, can be defined as follows:

```
//operation of isolating abstraction,

//implemented in Sympl as a function

function Aisol(x: logic): function;

begin

for i:=1 to length(x.P) do
return x.P[i] as function <P[i]>(_: logic): Boolean;

for j:=1 to length(x.Q).do
return x.Q[j] as function <Q[j]>( _,_: logic): Boolean;

end.
```

As a result of this function's call are returned a set of separate predicates (as an array of single argument functions) extracted from the array of properties *P* of the object *x*, and a set of separate relations (as an array of two argument functions) extracted from the array of relations *Q* of the same object *x*:

```
<P[1]>(_), <P[2]>(_),...,<P[n]>(_);

<Q[1]>(_,_),<Q[2]>(_,_),...,<Q[m]>(_,_).
```

Each element of the first array is a single parameter (argument) function (that looks like <Name> (_)), and each element of the second array is a two-place function (of the form <Name> (_, _)), where the symbol '_' denotes the names of the parameters of these functions that can be any, but of the logic type.

The resulting value of these functions when applied to a logic objects will be true, if a logic object/objects possesses/possess the property/relation or false if not.

If one executes this *Aisol* operation applying it to some logical object *x*, without returning the result to any variable, i.e. if one executes the code fragment "*Aisol (x)*", the results of such an operation will be available in the system as independently existing (free) predicates (functions) that can be applied to logical objects within the framework of the session.

These functions can be accessed by name, and not necessarily through the *P* or *Q* arrays' indexes as bounded predicates.

Another possibility realized in *Sympl* is to return the result of the execution of the *Aisol*() function to appropriate variables. These variables must be arrays:

```
(P: array[ ], Q: array[ ]):=Aisol(x).
```

The names of these arrays can be any, not necessarily *P* and *Q*, i.e. the coincidence or non-coincidence with the names of the property and the relations arrays *P* and *Q* of logical objects plays no role. Only the sequence of results plays a role: the set of separate predicates extracted from the *x.P* array is returned to the first array, and the set of separate relations from the *x.Q* array is returned to the second.

In this case, each of the newly received array elements is a function (its type being *function*), and not a string (its type not being *string*) as in the original arrays.

In the example above, the *P* and *Q* arrays are declared only when the *Aisol()* function is called (inside the parentheses), which indicates the receipt of the combined result (but not the union of the results).

In the framework of permissible syntactic polymorphism, the following possibilities of receiving results are also admissible in *Sympl*:

```
Aisol (x: logic; var P: array[ ], Q: array[ ]).

Aisol (x: logic) (P: array[ ], Q: array[ ]).
```

In the first case, the variables to which the *Aisol()* function should return the result are declared using the keyword *var*, and in the second, using an additional parameter list.

The variables *P[]* and *Q[]* can also be declared beforehand, before calling the *Aisol()* function:

```
P, Q: array of function.
```

Or like this:

```
P: array[ ].

Q: array[ ].
```

In the first case, *P* and *Q* are arrays of the type function, and in the second case are arrays of an indeterminate type.

In the second case the type casting for both arrays will be performed by the system when the *Aisol()* result are transferred to them.

The operation of identifying abstraction () of array elements by *P*(*x*), i.e. by predicates can be notated as:

```
function AequalP(X: array of logic): array of set;
begin
return Group X by P(X);
end.
```

The operation of identifying abstraction by Q (x, y) will be implemented as follows:

```
function AequalQ(X: array of logic): array of set;
begin
return Group X[ ] by X[ ].Q[ ] ;
end.
```

The *Group* ... *by* ... syntagma is a symbol solver's control command and is a function for obtaining combinatorial solutions.

The expression *Group X*[] *by X*[].*P*[] is a command that tells the system to group all the elements from the array *X* so that in each group there have to be at least two elements from *X*, each with at least one property from the array *P*, these properties being equal.

Literally this expression means the following: "group all the elements of the array *X* by the elements of the array *P*, belonging to the elements of the array *X*".

As a result of this command's fulfillment an array of equalization sets, that represent a structure and a remainder are returned. The array of identification sets looks like:

[{[X [i_1], X [k_1], X [l_1],...], [P_{a_1}, P_{b_1}, P_{c_1},...]},

{[X [i_2], X [k_2], X [l_2],...], [P_{a_2}, P_{b_2}, P_{c_2},...]},

...

{[X [i_n], X [k_n], X [l_n],...], [P_{a_n}, P_{b_n}, P_{c_n},...]}].

And the remainder, consisting of the elements of the array *X*, that don't have any common properties, i.e. cannot be equalized and make an identification set looks like:

[X[j], X[m], X[n], ...].

One can obtain this result using different ways of this command record, admissible within the framework of syntactic polymorphism as well:

1) Group X[i] by X[i].P[ ] where i:=1 to length (X);
2) Group X[i] by X[i]. P[j] where i:=1 to length (X), j:=1 to length (X[i].P);
3) Group X[i] by X [i]. P[j] where (i:=1 to length (X)) and (j:=1 to length (X[i].P));
4) Group X[ ] by X[ ].P[j] where j:=1 to length (x[ ].p).

In the above expressions, we used the "*where*…" supplement to the "*Group* ... *by* ..." syntagma to explicitly indicate the range of changes in the numbers (indices) of the array elements to be grouped.

The expression *X* [], which is a way to encompass all elements of the array *X*, is equivalent to the expression *X* [*i*], when the range of variation of *i* varies from the 1st to the last index, i.e. the length of the array. We used this property of the command's record in the expressions 1-3. In a similar way the range of variation of *j* index is indicated in the expressions 2–4.

It should be noted that such a way of recording this command excludes an erroneous access to nonexistent elements of the arrays *X* or *P* when the length of the arrays *X* or *P* equals to nil, as after the "where…" syntagma the range of index changes is clearly indicated, and no control over the access to nonexistent elements (as in the expressions *X*[] or *P*[]) is required, because in constructions of this type, it is assumed by default that the established operations apply only to arrays of nonzero length.

The expression *Group X* [] *by X* [] .*Q* [] indicates to the system that it is necessary to group all the elements of the array *X* into groups so that each group would have at least two elements *X* [] with at least one identical relation from the array of relations *Q*.

As a result of its execution, an array of identification sets is returned, which is a structure that look like:

[{[X [i_1], X [k_1], X [l_1],...], [Qa_1, Qb_1, Qc_1,...]},

{ X [i_2], X [k_2], X [l_2],...], [Qa_2, Qb_2, Qc_2,...]},

...

{[X [i_n], X [k_n], X [l_n],...], [Qa_n, Qb_n, Qc_n,...]}].

And a remainder of the elements of the array *X* that do not have any common relationships, i.e. are not identifiable:

X [j], X [m], X [n],...].

The identification set is a new data type and is defined using the following inference rules:

Set_of_Identification → '{['X ',' '[' Y ']';
X → xX x ',' xX| ']';
Y→ yY | y ',' yY| ']'.

The set of identification, as follows from its definition, consists of two parts, two arrays. The elements of the set are grouped on the left side, and the predicates of the set (properties or relations) on the right side.

Elements of the left side, i.e. the elements of the set proper, can be accessible in the following ways:

```
// access to the 1st element of the set

S {[1] |};

// select a random element of the set

select S;
```

Elements of the right side, i.e., the predicates of the set, can be accessible as:

```
// access to the 1st predicate of the set
S {| [1]};
// access to the i-th predicate of the set
S {| [i]};
```

Other operations applicable to the set are:

operation for including an element in a set. Returns true if the item has any common predicate with set data and was successfully included, returns false otherwise.

```
include (x, S);
```

operation for excluding an element from the set. Returns true if the element to be excluded belongs to the set. It is possible to exclude an element both by name and by index:

```
exclude (S, x);
exclude (S, i);
```

The operations of abstraction together with the operation of generalization are the basic operations of intelligence. The results of these logical operations are various concepts and categories.

A concept is a logical object that is a reflection of other logical objects and contains their distinguishing features. These include properties and relationships.

Each concept distinguishes between its content and scope. The content of a concept is the totality (set) of the attributes of objects reflected in it [2].

The scope of a concept is a set of logical objects, each of which has attributes related to the content of the concept, i.e. predicates.

Thus, a concept is a logical object that consists of two parts.

The concepts are the result of applying the abstraction of identification and are declared in *Sympl* as follows:

```
declare type notion: as record of
//notion's name
name: string;
// array of logical objects forming a set
X: array of logic;
// array of predicates (properties / relations) that all objects have
P: array of predicate;
end;
```

The objects displayed in the concept stand out from a composition of a broader set than the scope of the concept. Thus, we can talk about a hierarchy of concepts.

By the nature of the elements of their scope, concepts can be divided into non-collective and collective. Non-collective are those whose content features are proper to each scope element.

Collective are concepts in which the features that make up the content are inherent to the scope as a whole, rather than to its individual elements. For example, the constellation "Ursa major", "the collective of our institution." Extremely broad concepts are called categories.

As a rule, categories are philosophical concepts, for example, such as “matter”, “consciousness”, “being”, “essence”, etc. Categories can also be obtained by applying the classification component of a certain relation (or system of properties) as a generalizing one to other relations (or property systems). Moreover, words expressing specific concepts take additional generalizing meanings or new words are created for expressing these notions.

If one can move up the hierarchy of concepts (from categories and concepts of a lower level to concepts of a higher level) using the abstraction of

identification by adding new attributes, one can move (advance) in the opposite direction using the abstraction of identification by discarding existing ones.

The predicate type is defined in *SympI* as follows:

```
declare type predicate as function _(x: logic) or function _(x, y: logic);
```

This record means that a predicate type variable can be a single-function (record "*function _(x: logic)*") or ("*or*") a two-place function (record "*function _ (x, y: logic)*").

The underscore symbol '_' before the list of the function's parameters is a substitute for the function name.

Now, the initialization of a variable *notion1* of the type notion (i.e. a concept) will look like this:

```
// variable declarations
notion1: notion;
// declaration of an array of identification sets
M: array of set;
//initialization
// we apply to the variable M as the result of //the operation of identification by properties
M:=AequalP (X);
// if not an empty result
if not (M=void) then
begin
 // initialize the array of logical objects that
 //make up the concept
notion1.x: =M [1] {[] | };
 // initialize the array of predicates that all
// these objects have
notion1.P: =M [1] {\[]};
end.
```

As a result, we have an object of type *notion* (conceptual type), which is the result of applying the logical operation of identification abstraction implemented using the *Group* ... *by* ... syntagma and is a first-level abstraction.

To obtain second-level abstraction, it is necessary to apply the operation of identification abstraction to an array of logical objects *notion1.X* or to *M* [1] {[] | } and to some other array of logical objects that are the result of application of the abstraction of identification and are also logical objects that form the concepts of the first level.

One must note, that it is possible to apply the operation of identifying abstraction to concepts of different levels (for example, when one concept is an abstraction of the first level and the other is a concept of the second level abstraction), therefore it would be more correct to talk about the concepts of the first level and the concepts of a high level.

Let's consider an example of the operation of abstraction of identification by relation:

```
// declare an array of concepts
notion_array: array of notion.
// declare an array of identification sets
M: array of set;
//initialization
// assign the result of the operation of identification by relations to the variable M
M: = AequalQ(X);
// if not an empty result
if not (M=void) then
begin
// initialize the array in a loop
for i=1 to length (M) do
begin
notion_array [i] .X: =M [i] {[] | };
notion_array [i] .P: =M [i] {\ []};
end;
end;
```

The naming of the concepts obtained as a result of applying the abstraction of identification in *SympI* is yet an unsolved problem, though one can of course use an ordinal system of denotation, i.e. denote the first obtained concept as *C1*, the second as *C2*, etc., though in a natural language one creates new words for them, as well as the formalization of the abstraction of generalization.

For example, what do have in common two subjects "John" and "Jane" in the following two sentences: 1)"John helps poor people with money"; 2) "Jane volunteered to care for cleaning old people's flats"? If one will ask this question from any person, one will obtain the answer "that both are good (or good-hearted) people".

This is based on our picture of world as in it: 1) "helping people with money" is "good"; 2) "cleaning old people's flats" is "good" as well. But how can one obtain such an answer based on a logical approach?

The answer is that one must analyze a definite amount of texts, in which will occur a phrase that both activities are "good", i.e. one must find for both predicates "help people with money" and "clean old people's flats" common properties, that will be linked with the property "being good" in some or other way.

Thus, generalization is possible for two predicates when there exists an intermediate

predicate that can derivate from them directly or through a sequence of derivatives.

One can formulate this condition for properties (3), as well as for relations (4), in such a way:

$$A_{GENER}(P_1(x),P_2(y))=P_i: A_{ISOL}(P_1(x)) \rightarrow P_3 ... \rightarrow P_i \text{ U } A_{ISOL}(P_2(y))\text{->}...\text{->}P_i \quad (3)$$

$$A_{GENER}(Q_1(x_1,y_2),Q_2(y_1,x_3))=Q_i: A_{ISOL}(Q_1(x_1,y_2) \rightarrow Q_3 ... \rightarrow Q_i \text{ U } A_{ISOL}(Q_2(y_1,x_3)\text{->}Q_k ...\text{->} Q_i \quad (4)$$

Finding such intermediates in a text for two predicates is, however, not always possible. One must use text corpora, create semantic fields, but even in such a way the process will need human intervention. Thus, the problem of formalization of the abstraction of generalization remains unresolved.

5 Examples of the Proposed Theory Application

One can cite following examples.

Example 1. Automated functional entities hierarchy building

Let us consider a text, Text1 = “There is a bus going from the campus to the university. There is a plane that flies each Sunday from Moscow to Mexico”.

These sentences describe various entities, represented by words that are substantives/nouns, such as “a bus”, “a plane”.

By applying a morphological analyzer to the text we will obtain a tagged number of words for each sentence = {“There”=(“there”, Adverb), “is” =(“be”, Verb, Present Simple, 3rd person, indicative), “a”=(“a”, Indefinite Article), “bus”=(“bus”, Noun, Singular), “going”=(“go”, Verb, Present Continuous, person=any), “from”=(“from”, Preposition), “the”=(“the”, Definite Article), “campus”=(“campus”, Noun, Singular), “to”= (“to”, Preposition), “the”=(“the”, Definite Article), “university”=(“university”, Noun, Singular); “There”=(“there”, Adverb), “is” =(“be”, Verb, Present Simple, 3rd person, indicative), “a”=(“a”, Indefinite Article), “plane”=(“plane”, Noun, Singular), “that”= (“that”, Definite Article | Conjunction | Adverb | Pronoun | Adjective), “flies”=(“fly”, Verb, Present Simple, 3rd person), “from”=(“from”, Preposition), “Moscow”=(“Moscow”, Noun: Geographic Name, Singular), “to”= (“to”, Preposition), “Mexico”=(“Mexico”, Noun: Geographical Name, Singular), “each”=(“each”, Determiner | Pronoun), “Sunday”=(“Sunday, Noun: Time, Singular)}.

The syntactical structure of each sentence will look like:

1 (There is [=be]) & (a bus) & (going [=go] (from (the campus) to (the university)).
2 2. (There is [=be]) & ((a plane) & that & ((fly)& (each Sunday)& (from & Moscow) (to & Mexico)).

The logical structure of each sentence after the application of a logical analyzer will look like:

1 $\exists$ x_1= “bus”, x_2 = “campus”, x_3 = “university”: $x_1.P_1$= “go”: $x_1.P_1.P_1$=(name=“time”, value= “Present Continuous”), $x_1.P_1.Q_1$= “from”(x_2), $y_1.Q_2$ = “to”(x_3);
2 $\exists$ x_4= “plane”, x_5 = “Moscow”, x_6 = “Mexico”: $x_4.P_1$= “fly”; $\exists$ y_2 =$x_4.P_1$: $y_2.P_1$=(name=“time”, value=“each Sunday”), $y_2.Q_1$= “from”(x_5), $y_2.Q_2$ = “to”(x_6);

As the properties “go” and “fly” are complex ones and themselves possess properties and relations – they need to be decomposed to avoid hierarchical structures that will complicate their algorithmic processing:

1 $\exists$ x_1= “bus”, x_2 = “campus”, x_3 = “university”: $x_1.P_1$= “go”; $\exists$ y_1 =$x_1.P_1$: $y_1.P_1$=(name=“time”, value= “Present Continuous”), $y_1.Q_1$= “from”(x_2), $y_1.Q_2$ = “to”(x_3);
2 $\exists$ x_4= “plane”, x_5 = “Moscow”, x_6 = “Mexico”: $x_4.P_1$= “fly”; $\exists$ y_2 =$x_4.P_1$: $y_2.P_1$=(name=“time”, value=“each Sunday”), $y_2.Q_1$= “from”(x_5), $y_2.Q_2$ = “to”(x_6);

One must also introduce a logic rule that regulates the relations between bound predicates and the independent logic objects with the same names that are used for their full scaled definition in case when they are complex ones and have their one properties and relations.

if ($P_i \in x.P$, $P_k \in x.P$) $\cap$ ($\exists$ y_j: y_j.name = P_i, y_h: y_h.name = P_k) $\cap$ ($y_i \in \Re$) $\cap$ ($y_h \in \Re$) then ($P_i \in \Re$)$\cap$ ($P_k \in \Re$);

or

if ($P_i \in$ x.P, $P_k \in$ x.P) $\cap$ ($\exists$ y_j: y_j.name = P_i, y_h: y_h.name = P_k) $\cap$ Aequal(y_j, y_h) = true then Aequal (P_i, P_k) = true.

The programming implementation of this logical structure using Sympl is given below:

```
var
Propertyi, Propertyk: string;
Properties: set;
yi, yh: logic;
Rule: logicRule = <if exists(Propertyi: string) and
exists(Propertyk: string) and (Propertyi in Properties) and
(Propertyk in Properties) and (yi.name=Propertyi) and
(yh.name=Propertyk) then Aequal(Propertyi,
Propertyk).results:=Aequal(yi,yh).results)>;
Rule.Apply();
var
X: array of logic = (<"bus">, <"campus">, <"university">,
<"plane">, <"Moscow">, <"Mexico">);
X[1].P[1]=("go");
X[4].P[1]=("fly");
//declare an array of 2 logical objects y1=x1.P1= //"go" and
y2=x4.P1= "fly";
declare Y: array of logic = (X[1].P[1], X[4].P[1]);
Y[1].Q[1]=(<"from">(x:=X[2]), <"to">(y:=X[3]));
Y[2].P[1]=(name= "time", value="each Sunday");
Y[2].Q[1]=(<"from">(x:=X[4]), "to" (y=X[5]));
```

Let us apply now the operation of the abstraction of equalization by properties to the array of logical objects *X*.

```
notions: array of notion;
// declaration of an array of identification sets
M : array of set;
//initialization
// we apply to the variable M as the result of //the operation of
identification by properties
M:=AequalQ(Y);
// if not an empty result
if not (M=void) then
begin
l:= length(M);
for i:=1 to l do
 begin
 // initialize the array of logical objects that
 //make up the concept
  notions[i].X: =M [i] {[] | };
 // initialize the array of predicates that all
// these objects have
  notions[i].Pr: =M [i] {\[]};
 end;
end.
```

The array of identification sets obtained as a result of this operation will look like:

[{[y_1= "go", y_2= "fly"], [y1$_1$.Q_1= "from"(x)"to"(y), y_2.Q_1= "from"(x)"to"(y)]},

with a void remainder [].

The equalization between the properties x_1.P_1="go" x_4.P_1="fly was made according to their association to the logic objects y_1 and y_2 by names and the possibility of equalization of these latter objects by the relations y_1.Q_1= "from"(x_2), y_1.Q_2 = "to"(x_3) and y_2.Q_1= "from"(x_5), y_2.Q_2 = "to"(x_6).

In terms of the presented theory the result obtained in *notion[1]* corresponds to the concept "move". And one can name this notion manually:
notion[1],name:= "move";

The application of the operation of isolation *Aisol()* to *notions[1] Aisol(notions[1])* will give a sort of a prototype function - an analogue of a virtual function in Object Oriented Programming.

The application of the abstraction of isolation to the object *vehicle Aisol*() will return a logic function *move_in_space*(*x*: *logic*): boolean.

One can create a separate set of identification with the same name ("move") and a logic object that will be associated with this notion and function:
<"move_in_space">: set:=notions[1].X;
<"move_in_space">: logic;
move_in_space.Q[1]:=from(x: logic) to (y: logic).
The association is done automatically by identical names.

The body of this function can be implemented as:

```
move_in_space(x: logic).body = begin
 for i=1 to length (x.P) do
  begin
   if  x.P[i] in move_in_space then
    begin
     move_in_space:= true;
     break;
    end;
 end;
```

Example 2. Automated object entities hierarchy building

Let us now apply in the scope of the programming code of the Example 1 the operation of equalization by properties to the logical objects of the array *X*:

```
M: array of set;
M:=AequaPl(X);
    // if not an empty result
    if not (M=void) then
    begin
    l:= length(M);
    for i:=1 to l do
     begin
```

```
 // initialize the array of logical objects that
 //make up the concept
  notions[i].X: =M [i] {[] | };
 // initialize the array of predicates that all
// these objects have
  notions[i].Pr: =M [i] {|[]};
 end;
end.
```

The array of identification sets obtain as a result of this operation will look like:

[{[x_1= “bus”, x_4= “plane”], [$x_1.P_1$= “go”, $x_4.P_1$= “fly”]}],

and the remainder will look like [x_2, x_3].

In terms of the presented theory the result in the *notions[1]* corresponds to the concept “vehicle” as “A vehicle is an object that can move in space”, i.e. “from the point x to a point y” and will be some sort of analogue of an object prototype that will correspond to a parent class in OOP:

```
notions[1].name:=“vehicle”;
<“vehicle”>: set:=M[1]{[]|};
<“vehicle”>: logic
( P= (“move_in_space”);
  Q = ();
).
```

The declared above logic object “vehicle” has a single element of the properties array P[1]= “move in space” and a void array of relations.

6 Conclusion

The presented abstraction theory and its algorithmic implementation as was shown in the examples can be used primary for obtaining new knowledge patterns.

Following tools have been developed in the paper: a type “*logic*” (logical data type), which implements a logical object with a set of properties and relations implemented as arrays; *Aisol()* function that implements the logical operation of isolating abstraction; *AequalP()* function that implements the logical operation of property abstraction; *AequalQ*() function that implements the logical operation of abstraction of identification over relations; the *Group* ... *by* ... syntagma, which allows to group logical objects by their properties or relationships using a symbol solver built into the system and obtain an array of identification sets as a result; a type “*set*” (a type that implements the identification set), which is a set on the left side of which is an array of logical objects that are elements of this set, and on the right side of which is an array of predicates (properties or relations) that are common for all of its elements; the operation of accessing an array of elements of the identification set: *M{[]}*; the operation of accessing the array of predicates of the identification set: *M{|[]}*; the operation of selecting a random element from a set: *select*; the operation of including an element into a set: *include*; the operation of excluding an element *x* from the set: *exclude*; the operation of excluding an element from the set by its index: *exclude*; the type “*notion”* (“conceptual data type”).

The area of application of the presented theory is an automatized building of text entities and based on them automatized declaration of class hierarchy and virtual functions.

As a result of further development in this direction, it is also possible to implement such logical concepts as category and the category’s data type (“categorical data type”).

However, the problem of formalization of the abstraction of generalization remains unresolved.

Acknowledgments

Dedicated to the author´s father, Professor Valerian Zheltov. This paper has been supported by the Kazan Federal University Strategic Academic Leadership Program ("PRIORITY-2030"), Strategic Project #4.

References

1. **Welling, H. (2007).** Four mental operations in creative cognition: The importance of abstraction. Creativity Research Journal, Vol. 19, No. 2-3, pp. 163–177. DOI: 10.1080/10400410701397214.
2. **Gorskiy, D. P., Ivin, A. A., Nikiforov, A. L. (1991).** Kratkiy slovar' po logike [A Concise Dictionary of Logic], Moskva Prosveshchenie.
3. **Tennant, N. (2004).** A general theory of abstraction operators. The Philosophical Quarterly, Vol. 54, No. 214, pp. 105–133. DOI: 10.1111/j.0031-8094.2004.00344.x.
4. **Silbermann, F. S. K., Jayaraman, B. (1989).** Set abstraction in functional and logic programming. Proceedings of the Fourth International Conference

on Functional Programming Languages and Computer Architecture, pp. 313–326. DOI: 10.1145/99370.99398.

5. **Yokomori, T. (1987).** Set abstraction-an extension of all solutions predicate in logic programming language. New Generation Computing, Vol. 5, pp. 227–248. DOI: 10.1007/BF03037464.

Article received on 02/12/2022; accepted on 20/03/2023. Corresponding author is Pavel Zheltov.