Mathematical Problems of Computer Science 39, 54--65, 2013.

A Combinative Approach to Generalization of Meanings

Karen S. Khachatryan

State Engineering University of Armenia
e-mail: khkarens@gmail.com

Abstract

We gain new meanings through the acquisition of communities, the revelation from experience and the creation of new meanings by combining the existing ones. We refine meanings by abstract classes combined by be-, have-, do- categories of relationships [1, 3, 7]. In the paper we present a novel combinative algorithm constructing new meanings by generalization of the given sets of meanings.

Keywords: meanings, meaning processing, combinative generalization.

1. Introduction

One way of new meaning extraction is the combination of available ones. Particularly, the computation of a (or *the*) least generalization of two or more meanings helps to build a meaning expressing their general properties. It is a fundamental problem in inductive inference occurring particularly in machine learning [2, 6]. It may help to start from a set of descriptions assumed to be examples of the same meaning and consider their least generalization as a working basis. The operation can also be used to organize a large set of descriptions in a hierarchical structure.

Generalization and specialization problems have been studied for different knowledge representation models. [6] summarizes and distinguishes the methods of generalization into two categories: a generalization by features and a structural logical (or conceptual) generalization. The first methods usually solve the problems of classification and formation of meanings. [6, 13, 15] review and analyze the methods of generalization by features. These include a *method of potential function* as a function ψ_k for the class *K* which is built to have the maximal value on the set of objects $s \in K$ [16]; and *a generalization by features* using either *the voting method* proposed in [13, 14] or *the covering technique* [17]: first it prepares the input training data for building of base classifiers by perturbing the original training data, and builds base classifiers on the perturbed data, then the proximity measure is determined for the object *s* to each of the classes K_j by comparing the values of features of the object *s* from the specified subsets with the appropriate values of features of etalon objects.

A wide set of generalization/specialization techniques have also been developed for conceptual graphs which are well studied language of knowledge representation and reasoning [2, 10]. These include techniques like the evaluation of *the least upper bound (the least generalization) H* as an irredundant form of the categorical product of two basic conceptual graphs (BG) *B* and *G:* $H = B \times G$; the *maximum join operation* between two BGs *G* and *H* as a composition of the following steps: first it merges a concept node in *G* and a concept node in *H*,

and then continues merging as far as possible neighbors of previously merged nodes [12]; and the *extended join operation* which generalizes maximal join operations by using the properties of compatible partitions of the concept node set of a BG [11].

In the Solver of SSRGT class problems (problems, where Space of Solutions can be represented by Reproducible Game Trees) [1, 3, 7], the task of finding *the least generalization* in its basic form takes two meanings, say *A* and *B*, and asks for a least generalization of *A* and *B*, i.e., a meaning *K* such that $K \succcurlyeq A$ (*K* is a generalization of *A*) and $K \succcurlyeq B$ and for all meanings *K'*, if $K' \succcurlyeq A$ and $K' \succcurlyeq B$ then $K' \succcurlyeq K$. In other words, given two acquired meanings *A* and *B*, extract a new meaning *K* which will represent the general characteristics of both *A* and *B*, while taking as a criterion of the generalization the most specific meaning which can be extracted.

Further in the paper we give the definition of the *skeleton of a meaning* as a sub-graph which is necessary to compose and correctly activate the meaning from sub-meanings [5] and define the elementary generalization and specialization operations for it. Next we define the *generalization* (specialization) as a sequence of elementary generalization (specialization) operations and finally propose an algorithm of evaluating a least generalization of two meanings. We represent the strategy of selecting the generalizable sub-meanings from two meanings and detail the generalization procedure. We show that the algorithm is able to

- *find common parts of two meanings*, i.e., if there are meanings "Two Pawns" and "Two Knights", then it extracts a new meaning "Two Figures" etc.,

- *dynamically generate and integrate a new meaning between the be connection chain,* for example, it extracts and integrates "FieldUnderCheckOfPawn" new meaning between "FieldUnderCheck", "FieldUnderCheckOfPawnAtPos1" and "FieldUnderCheckOfPawnAtPos2" meanings,

- *extract a common Interface (pattern)*, the algorithm is able to extract "FieldUnderCheck" meaning by analyzing the list of "FieldUnderCheck of Specific Figure" meanings.

In the conclusion we summarize the main findings of the research.

2. The Skeleton of a Meaning

[3] discusses the means of meanings acquisition and the algorithm of their integration into the internal graph of abstracts. Within the graph, each meaning is represented as a sub-graph centered in the meaning node and having other nodes as sub-meaning while edges as one of *be-*, *have-* or *do* connections [1, 3, 5]. Note that all edges are bidirectional, in other words, if there is a *be* connection between the nodes *A* and *B* indicating that *A* is a *base type* of *(subsumes) B* then also a reverse connection is built between *B* and *A* indicating that *B* is a *sub type* of *(subsumed by) A*. Similarly, the reversed connections are built also for *have* and *do* edges. Therefore, when referring to these bidirectional edges, we distinguish the roles for nodes as *sources* and *destinations.* We call a node *source* for *be* connection if it is the *sub type,* consequently, the *base type* becomes a *destination.* Similarly, *a source* for *have* connection is the node that has the other node as an attribute. And, finally, for *do* connection the precondition node serves as a *source* while the action itself becomes a *destination.* What follows is that the above definition of meanings is quite wide and assuming that all meanings are using the same set of nucleus meanings and taking the allowed distance of a sub-meaning from the central node as big as we want, we can end up having almost the whole connected component (even a graph of abstracts itself) as a representation of a single meaning (regardless what meaning we pick). Therefore, we define the *skeleton of a meaning* as a sub-graph which is necessary to compose and correctly activate [5] the meaning from sub-meanings. It is constructed from the meaning's graph by recursively traversing only the following set of edges and connected nodes:

1. If the node is a *nucleus* then stop further processing.

2. If the node is *neither a virtual nor a usage* then as a next layer of the skeleton select only *Have* connections where the role of the node is a *source*. This means, that none of *Be*, *Do* connections or *Have* connections, where the node is a *destination*, will be considered.
3. If the node is *a virtual or a usage* then only *Be* edges, where the node is a *destination*, must be selected for further processing. For virtual nodes, this includes the set of all specifications of the node. For the usage nodes, this is basically the base virtual node (it is being selected because of the reverse *Be* connection)

And, finally, applying steps 1 to 3 recursively on each new node will result in a complete skeleton of the meaning.

3. Elementary Generalization/Specialization Operations

We consider the generalization of meanings as a generalization of their skeletons. Hereafter saying a meaning we refer to the skeleton of a meaning: we will use an *entire meaning* notation to refer to the meaning with the original definition.

Let us describe the generalization of a single meaning before considering the computation of a least generalization of two or more meanings. A partial order is interpreted as a generalization or a relation: $m_1 \geqslant m_2$ means that the meaning m_1 is a generalization of the meaning m_2 (or m_2 is a specialization of m_1, m_1 subsumes m_2 or every entry having a meaning m_2 has also a meaning m_1).

A generalization is a "unary" operation, i.e. it has a meaning as an input and a meaning as an output. We define the following elementary generalization operations:

Increase. Increase the type of a sub-meaning. More precisely, given a meaning *A*, a sub-meaning *x* of *A*, and a type $T \geqslant x$ *increase (A, x, T)* is the meaning obtained from the *A* by increasing the type of *x* up to *T*. Within our meaning representation model the *increase* operation means replacing the sub-meaning with one of the meanings in its *be* connection chain Fig. 1 a). Similarly, the *Increase* operation is defined for the relations of sub-meanings. Given a meaning *A* and a relation *r* for a sub-meaning *x* of *A*, and a relation $r \in R$, then *increase(A, x, r, R)* is the meaning obtained from the *A* by changing the relation *r* of *x* up to the relation *R*. The increase condition of a relation indicated that the new relation *R* should enclose the value range defined by the relation *r*.

Substract. Given a meaning *A*, and a set of sub-meanings $C_1, C_2, \dots, C_k$ of *A*, then $substract(A, C_1, C_2, \dots, C_k)$ is the meaning obtained from *A* by deleting $C_1, C_2, \dots, C_k$ sub meanings and any relation which has a reference to them (the result can be the empty meaning). Similarly, the *substract* operation for relations is defined as follows: given a meaning *A*, and a set of relations $r_1, r_2, \dots, r_m$ for the sub-meanings $C_1, C_2, \dots, C_k$, then $substract(A, C_1, C_2, \dots, C_k, r_1, r_2, \dots, r_m)$ is the meaning obtained from *A* by dropping the relations $r_1, r_2, \dots, r_m$ Fig. 1 b).

The elementary specialization operations are defined as inverse operations of the elementary generalization operations. They are as follows:

Restrict. Given a meaning *A*, a sub-meaning *x* of *A*, and a type $T \leqslant x$ *restrict(A, x, T)* is the meaning obtained from *A* by decreasing the type of *x* to *T*. Within our meaning representation model the *restrict* operation means replacing the sub-meaning with one of the meanings in its reverse *be* connection chain Fig. 1 c). Similarly, the *restrict* operation is defined for relations of sub-meanings. Given a meaning *A* and a relation *r* for a sub-meaning *x* of *A*, and a relation $R \in r$, then *restrict(A, x, r, R)* is the meaning obtained from the *A* by changing the relation *r* of *x* down to the relation *R*. The restrict operation indicated that the relation *r* should embrace the relation *R*.

Disjoint Sum. Given two disjoint meanings *A* and *B*, *A+B* is the union of *A* and *B* that is the meaning which has *A* and *B* as sub meanings. Similarly, for relations, a new relation *R* can be defined for sub meanings Fig. 1 c).

We define a *generalization/specialization* relation (or *subsumption*) by a sequence of elementary operations

Definition. A meaning *A* is a generalization of a meaning *B* if there is a sequence of meanings $A_0 = B, A_1, \dots, A_n = (A)$, and, for all $i = 1, \dots, n$, A_i is obtained from A_{i-1} by a generalization operation.

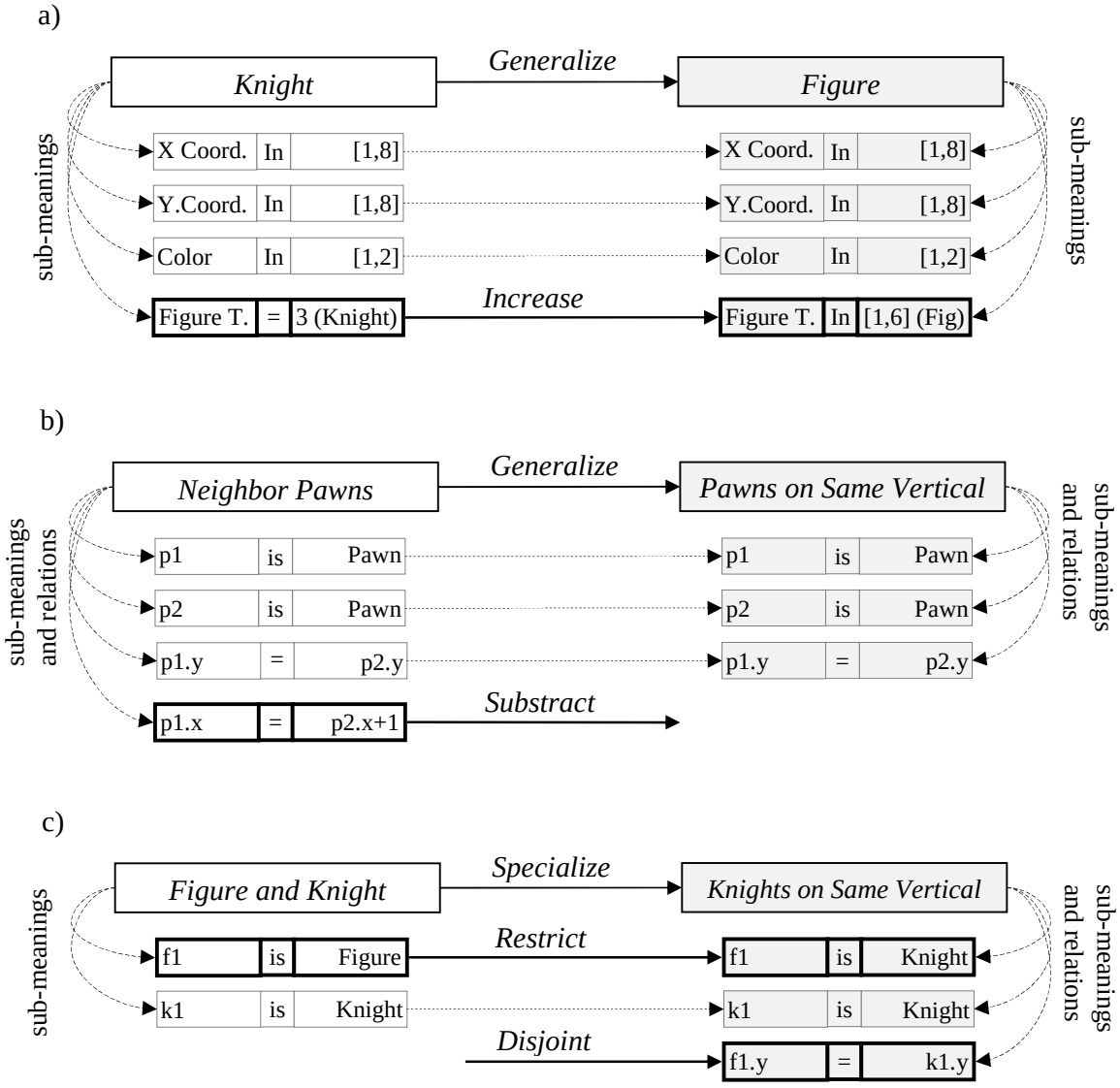

Fig. 1. Elementary generalization and specialization operations. a) Generalization of *Figure* meaning from *Knight* meaning by an elementary *Increase* operation applied on *Figure Type* sub-meaning. b) Generalization of *Pawns on Same Vertical* meaning from *Neighbor Pawns* by applying *Substract* elementary operation on $p1.x = p2.x + 1$ relations. c) Specialization of *Figure and Knight* meaning to *Knight on Same Vertical* by applying *Restrict* elementary specialization operation on *f1:Figure* sub-meaning and *Disjoint Sum* operation to add new relation between *f1* and *k1* sub-meanings: $f1.y = k1.y$.

From the definitions it follows that *B* can have extra sub-meanings and relations which don't exist in *A.*

4. The Generalization of Two Meanings

Finding a (or the) least generalization of two meanings *A* and *B* is to find a meaning *K* such that $K \succcurlyeq A$ and $K \succcurlyeq B$ and for all meanings *K'*, if $K' \succcurlyeq A$ and $K' \succcurlyeq B$, then $K' \succcurlyeq K$. In other words, the algorithm has to traverse both meanings (abstracts in the meaning graph) and compose a new abstract from the least generalizations of their sub-meaning pairs and relations. The latter one can be either memorized or dropped based on some post processing algorithm: for example, if we deal with interactive expert systems, then the system would rather ask the expert for further analysis. Alternately, it can maintain a weighted graph of abstracts and evaluate durables during the time [1], however, the consideration of the post-processing algorithms is out of the scope of this paper, therefore, we suppose that an expert can be asked to approve or reject the new evaluated meaning.

The paper aims to answer the following major questions which arise during the processing of the algorithm:

- how to select the pairs of sub-meanings (abstract's attributes) to be generalized into the new meanings?
- how to generalize them?
- how to verify/extract the relations (dependencies) between sub-meanings?

4.1. Select the Attribute Pairs to Generalize

In this section we will describe the algorithm which extracts the sets of attribute pairs to compose least generalizations form user/expert defined meanings. Note, that there can be more than one least generalization when considering both sub-meanings and relations. For example, let's suppose there are a meaning *A* and a meaning *B* where *A* is composed of sub-meanings $f_{1A}: Figure$, $p_{1A}: Pawn$ and $p_{2A}: Pawn$ (*t:T* notation means that *t* is a type of *T*) and there is a relation $f_{1A}.x = p_{1A}.x + 1$ defined for f_{1A} and p_{1A} attributes. On the other hand, *B* has two sub-meanings $p_{1B}: Pawn$ and $p_{2B}: Pawn$ with a relation $p_{1B}.x = p_{2B}.x + 1$. Let's also suppose that $Figure \succ Pawn$ ($Figure$ strictly subsumes $Pawn$). What follows is that the algorithm can either specialize the relation and extract a least generalization G_1 composed of two *Pawns*, or specialize the sub-meaning and extract a new meaning G_2 composed of *Figure* and *Pawn* by keeping the relation defined between them.

In order to find the compatible pairs of sub-meanings, let us recall the structure of graph of abstracts and the semantic connections existing between the abstracts (the semantics of *be, have* and *do* connections). As defined in [3], there are the following types of GA nodes: *nucleus* - smallest representation units of meanings, *ar1* - abstracts having only nucleus attributes (there can be at most one attribute of a given nucleus type), *sets* - representing a group of abstracts with similar characteristics, *composite abstracts* - a complex form of abstract representation, they can have any kind of attributes, *virtual abstracts* - composite abstracts with attributes having undefined relations, and *actions* - representing the action meanings. Considering these categories of GA nodes, as a first step, the algorithm segregates attributes into different compatible groups and evaluates the pair extraction between the groups containing nodes having the same type.

4.2. Generalization of Nucleus Abstracts

Property. Two nucleus abstracts are generalizable if and only if they have the same type.

The proof simply follows from the definition of nucleus abstracts: they represent different nucleus characteristics, hence, cannot be generalized further.

We shall recall the structure of a nucleus abstract: it has a single attribute and value range defined for that attribute. For the generalization we shall assume one of the following operations:

- *union of regulations*
- *finding the closest base type which covers both sets defined by regulations*.

Both of these have a practical sense, however, we first try to find a common base nucleus type by traversing their *be* connection chain in order to avoid the expansion of new nucleus abstracts by unifying different nucleus values (this can lead to a creation of new types for all possible combinations of nucleus values).

4.1.Generalization of AR1 Abstracts

According to the definition of ar1, it contains at most one element of a given nucleus type (Fig. 2). Therefore:

Fig. 2. A and B ar1s.

Property. The attributes of two ar1s can be generalized only if they have the same nucleus types, moreover, there is at most one possible pair for a given attribute.

Table 1 represents the mapping between *A* and *B* ar1s' attributes. As we see *A.r1*, *B.s2* and *B.p2* attributes do not have pairs, consequently, they are ignored in the generalized abstract and a new ar1 is constructed by generalizing the paired nucleus attributes ($x1{:}\,x2 \rightarrow x$, $y1{:}\,y2 \rightarrow y$ and $z1{:}\,z2 \rightarrow z$ in this example). The asset of ar1s is that there is no dependency defined between the attributes. The only dependency (belonging to the same id group) is implied in [3].

Table 1. The mapping table of AR1s attributes.

A	Nucleus Types	B
x1	*x*	x2
y1	*y*	y2
r1	*r*	-
z1	*z*	z2
-	*s*	s2
-	*p*	p2

4.2. Generalization of Set Abstracts and Actions

The generalization of Sets is done by generalizing the composite element of the Set and uniting the min-max ranges of source Sets.

Here *actions*' generalization is discussed only by their preconditions, which in their turn, are *composites.*

4.3. Generalization of Composite Abstracts

The least generalization of two meanings is, basically, the least generalization of two composite abstracts. Intuitively, the effect of the least generalization is to find the biggest subgraphs of two meanings which can be merged into one. In this section we will present some general ideas behind the operation and will give the algorithm adopted for Solver's meaning representation.

Definition. Let *A* and *B* be two disjoint meanings and *c* and *d* - two sub-meanings in *A* and *B*, respectively. A *generalization* of *A* and *B* is a least generalization of *c* and *d* sub-meanings.

A way of extending a *generalization* of *c* in *A* and *d* in *B* consists of searching the neighbors (sub-meanings connected with them through relations) of *c* and *d*, then to check if these nodes can be generalized and so on. In other words, starting from a pair of generalizable sub-meanings, the idea is to search, in a greedy way, generalizable neighbors of previously identified generalizable nodes. The resulted least generalization is, thus, locally "maximal".

In order to specify a *least generalization* operation, one has to define not only the conditions for generalization of sub-meanings and relations but also a *strategy* for exploring the meaning

graphs. Given two generalizable meaning nodes as a starting point, there may be several least generalizations, but computing one of them can be done in polynomial time, whereas computing ***the*** *least generalization with* a maximum number of nodes is NP-hard (indeed it admits the homomorphism or injective homomorphism as a special case) [2].

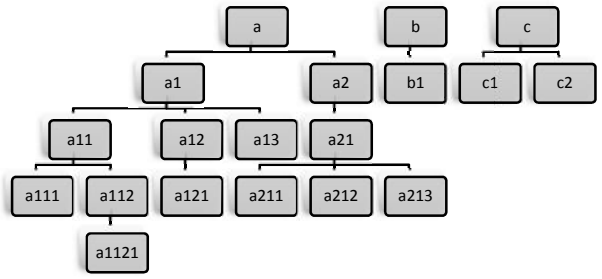

Fig. 3. The type hierarchy.

In order to improve a least generalization obtained by a greedy approach, we propose a strategy for picking the starting nodes and exploring meanings' graphs. In the strategy we are excessively using the structure of the meanings' graph and particularly, the semantics of *be* connections (as we did in ar1s).

Definition. Two abstracts are *strongly compatible* if they have a common node in the chain of their *be* connections.

The *be* connection is one of the major relations defined between the abstracts (during the acquisition procedure). Therefore the existence of a common base type indicates the importance of the connection between two abstracts. On the other hand, the lack of a common base type means that they had not strong connections during the acquisition procedure. It could also be possible that the post processing algorithm or an expert discarded evaluated generalizations for these abstracts, thus taking into account that the sub types have to satisfy also the restrictions defined in the base types, it can indicate that there is no acceptable generalization for these abstracts.

Property. The closest is the common node the *more strongly compatible* are abstracts.

Proof of the property follows from the semantics of *be* connection. In this connection there are two components, namely, *base* type and *sub* type. The *sub* type is constructed from the *base* type by inheriting from it and possibly adding more restrictions. However, any instance of the *sub* type will also satisfy the regularities defined in the *base* type, hence, the closest *base* type contains the most common characteristics.

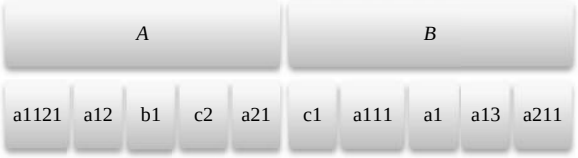

Fig. 4. A and B composite abstracts.

Using this property we propose a substructure for the meanings' graph exploration strategy to pick attribute pairs standing closest in the be connection chain.

Let's suppose we have the hierarchy

Table 2. The initial mapping list.

Left Container	List	Right Container
a1121,	*a1121*	-
a1121	*a112*	-
a1121	*a11*	a111
a1121, a12	*a1*	a111, a13, a1
a1121, a12, a21	*a*	a111,a211,a13, a1
a12	*a12*	-
b1	*b1*	-
b1	*b*	-
c2	*c2*	-
c2	*c*	c1
a21	*a21*	a211
a21	*a2*	a211
-	*c1*	c1
-	*a111*	a111
-	*a211*	a211
-	*a13*	a13

described in Fig. 3. The *Be* chain of the abstract *a1121* is: $a1121 \rightarrow a112 \rightarrow a11 \rightarrow a1 \rightarrow a$. This means that the top type is *a*, while the closest type is *a112*. Let's suppose we want to generalize abstracts *A* and *B* from Fig. 4.

Table 3. The mapping list after sorting.

Left Container	List	Right Container
a1121,	*a1121*	-
a1121	*a112*	-
c2	*c2*	-
a12	*a12*	-
-	*a111*	a111
-	*a211*	a211
-	*a13*	a13
b1	*b1*	-
b1	*b*	-
-	*c1*	c1
c2	*c*	c1
a21	*a21*	a211
a21	*a2*	a211
a1121	*a11*	a111
a1121, a12	*a1*	a111, a13, a1
a1121,a12, a21	*a*	a111,a211,a13, a1

We compose the substructure by the following rules. For the left abstract we initialize a list of all nodes which appear in the *be* connection chain of each attribute (including itself) and keep reflexive mapping from the attribute to the base types in a list. Note, that during the initialization, the most specific types for a given attribute are inserted first. If a base type with the similar id (name in our case) already exists in the list, then we simply increase the number of references to that element and add the pointing attribute to the element's left container.

Similarly, we iterate over all attributes of the right abstract and integrate all base types into the list with a difference that each attribute is added to the right container of the base type (Table 2). As a next step we count the number of elements in the left and right containers of base types and sort the list in the increasing order of the cumulative element count in right and left containers (Table 3). Meanwhile, if the number of elements is the same for two entries, we keep the initial ordering (see $a21 \rightarrow a21$ *and* $a21 \rightarrow a2$ *mapping* in Table 3). From the definition of the structure it follows that the base types appearing in the upper levels of the hierarchy will have a bigger number of connected elements. This is because each attribute from the bottom levels will increase also the number of connected elements of base types. Therefore, we can argue that mappings' entries appearing first for the given attribute are the closest base types and the later the mapping entry appears the further is the base type.

This leads us to the selection algorithm of the best matches of attributes in two abstract. We start iterating over the elements of the sorted list and remove the mapping entries which contain only right or left container (Table 4).

Table 4. The list without pairless elements.

Left Container	List	Right Container
c2	*c*	c1
a21	*a21*	a211
a21	*a2*	a211
a1121	*a11*	a111
a1121, a12	*a1*	a111, a13, a1
a1121,a12, a21	*a*	a111,a211,a13, a1

From the remaining set, if there is a one to one mapping (like $c2: c1 \rightarrow c$ or $a21: a211 \rightarrow a21$), then these attributes are paired and are removed from the lists of bottom entries (Table 5). If there is more than one attribute in one of

Table 5. The list after removing one to one mapping entries.

Left Container	List	Right Container
a12	*a1*	a13, a1
a12	*a*	a13, a1

containers then we have an uncertainty, therefore new sets of pairs are created for all the possible combination chains. For the given example, two set of pairs will be generalized:

$\{ c2: c1 \rightarrow c,\ a21: a211 \rightarrow a21,\ a1121: a111 \rightarrow a11, \boldsymbol{a12: a13 \rightarrow a1}\}$

$\{ c2: c1 \rightarrow c,\ a21: a211 \rightarrow a21,\ a1121: a111 \rightarrow a11, \boldsymbol{a12: a1 \rightarrow a1}\}$

As a result of this procedure we get a list of arrays of attribute pairs. Each element of the array represents one possible least generalization of two input abstracts. We shall note here that some of the attributes might be ignored because of not having proper pairs (*b1* in *A*, for example). The important achievement of this strategy is that we significantly reduced the number of possible least generalizations. It is *only* multiplied if there are attributes having the same types, but mainly, in the definition of a composite abstract the same attribute is not used multiple times (there are other types, like Sets to be used for such kind of definitions).

4.4. Generalize Paired Attributes

At this point for each array of paired attributes we have to extract their least generalizations and consider the extraction of relations. Finally, we have to compose a new abstract by putting together all these "building blocks".

First of all, let us discuss the approaches of extraction least generalization of paired attributes. The key point here is that they have an evaluated common subsumer (a common base type). Based on the type of the subsumer, i.e. whether it is a *virtual abstract*, thereby indicating the attributes being *usage* nodes [3], or not, the algorithm adopts different strategies. If the attributes are usages then the common subsumer is taken as a least generalization and only new relations are further analyzed for them. This is because the usage nodes do not add any additional attribute to the base type, rather, they only specify more restrictions. On the other hand, if the base type is not a virtual then the set of attributes, which exist in the base type, are extracted as a part of a generalization and the remaining ones are generalized further. More precisely, let D_1 and D_2 be paired attributes which have a common base type B which is not virtual. In this case the least generalization contains the set of attributes defined in *B* merged with the generalization of D_1/B and D_2/B.

An interesting property of the new generalized abstract from the pairs having the same base type is that it is either the base type itself or an abstract holding a place between the base type and sub types. In other words, from the property of *be* connection, it follows that there is a homomorphism from the common base type to the new generalized abstract. This action leads to the organization of a hierarchical structure between definitions. For example, by generalizing "FieldIsUnderCheckOfPawnPos1" and "FieldIsUnderCheckOfPawnPos2" meanings the algorithm can extract a new meaning "FieldIsUnderCheckOfPawn" and integrate it into the *be* connection chain between these meanings and their base type - "FieldInUnderCheck" meaning.

Once the paired attributes of each abstract have their mappings in the generalized one, the algorithm starts evaluating and extracting relations defined between the attributes and integrate them into the generalized abstract.

Fig. 5. Generalization of c1 and c2 paired abstracts.

A relation/regulation between an abstract's attributes is called a dependency and has $< attr >< ROP >< expr >$ form, where *attr* is the name of the attribute, *ROP* is a relational operator $(=, \neq, <, \leq, >, \geq)$ and *expr* is an arithmetic expression. Naturally, the generalization algorithm has to consider the existence and extract the analogical dependencies from two source abstracts. Thence, the algorithm has to analyze each dependency and check:

- is it still valid on the generalized abstract?

- is there an equivalent dependency in the pair abstract?

First it is necessary to ensure that none of the dependent attributes is excluded to continue the further verification. This could happen if abstracts contain extra attributes which are missing in the generalized one. For example, let us suppose *c*, *c1* and *c2* have the attributes given in Fig. 5. Moreover, $c1.p1 = c1.r + 2$, $c1.q1 = c1.p1 - 2$ and $c2.q2 = c2.p2 - 2$, $c2.s = c2.p2 - 3$ are the dependencies defined for them. Let us also suppose that the algorithm picked $c1: c2 \rightarrow c$ pair for the processing and *c* is picked as the generalized abstract. Thus, the dependencies $c1.p1 = c1.r + 2$ and $c2.s = c2.p2 - 3$ cannot be verified, because *s* and *r* attributes do not exist in the abstract *c*. Therefore, these dependencies are dropped. The other two are analized further. The second step is to perform the referred attribute name replacement in dependency expressions. To do that the experssion is parsed and an expression tree is built. Afterwards, each reference node is changed to point to the exact node in the generalized abstract: $c1.p1 \rightarrow c.p$, $c1.q1 \rightarrow c.q$, $c2.p2 \rightarrow c.p$, $c2.q2 \rightarrow c.q$, correspondingly.

The final step is to extract the equivalent expressions from the set of dependent expressions. Note, that we use the term equivalence instead of isomorphism because the free literals are the same within both expressions. To decide whether two arithmetic expressions are equivalent is an important problem in computational theory [9]. However, the general problem of equivalence checking, in digital computers, belongs to the NP Hard class of problems [8]. Even though, there are different algorithms which are fast enough to be used in practice. From that point of view we have adopted the algorithm proposed in [4]. Its technique is specifically designed to solve the problem of equivalence checking of arithmetic expressions obtained from high-level language descriptions, which consists of regular arithmetic operators (+, -, ×) and logical operators (and, or, not). The method uses *interval analysis* [10] to substantially prune the domain space of arithmetic expressions (and conditional expressions) and limit the evaluation effort to a sufficiently small number of minimally sized spaces within the domain of the expression. Then, it is extended to the technique to incorporate the arbitrary use of logic operators *and*, *or*, and *not* within the arithmetic expressions.

Thus, applying the above technique equivalent dependencies are extracted and integrated into the generalized abstract.

4.5. Extraction of a Virtual Abstract

The difference between virtual and non virtual composite abstracts is that the first one has attributes with undefined relations. Here we will discuss the extraction of a new virtual abstract by generalizing both from virtual and non virtual abstracts. The major difference of the procedure, compared to the procedure of extraction of composites, is the handling of the regulations when generalizing paired attributes. Thus, the initial steps of finding the attribute pairs and the algorithm of verifying/extracting the relations between the attributes, are replicating the ones defined for composite abstracts, however, the handling of regulations are defined for an attribute in the topmost, nucleus, level drags in the peculiarities. Here, if regulations differ then rather than applying *Increase* elementary operation (up to a *"*"* symbol representing all applicable values) we replace them with a symbol representing an undefined relation: *"?"*. This leads to a core difference between two generalizations. The first one is like a complete class and can be used to instantiate an object. However, the property of undefined relation drives the virtual generalization closer to the Interfaces or abstract classes in OOP languages (Java, etc.). In other words, as a result of a generalization the algorithm is capable of extracting categorically new abstracts. For example it is possible to extract "FieldUnderCheck" virtual abstracts by generalizing "FieldUnderCheckOfKnightPositionPattern1" and "FieldUnderCheckOfQueen PositionPattern1" abstracts.

5. Conclusion

In the paper we discussed the generalization and specialization operations for meanings within the be-, have-, do- linguistic representation model of SSRGT Solver. Particularly we defined *Increase and Substract (Restrict* and *Disjoint Sum)* elementary operations and *generalization (specialization)* operation by means of their sequence.

Next we considered a (or the) least generalization of two acquired meanings and proposed a strategy for selecting the attribute (sub-meaning) pairs to be generalized into the new meaning. Moreover, we claimed that the attributes pairs selected by our algorithm are *strongly compatible*. In other words, they are the closest pairs in the *be* connection chain. In order to deal with the extraction of relations/dependencies between attributes we adopted the algorithm of equivalence checking of arithmetic expressions from [4].

Further we showed how the generalized meanings were integrated into the meaning hierarchy and represented the algorithm of extraction of *virtual* generalizations. Our experiments showed that the proposed algorithms were able to:

- find common parts of two meanings,
- dynamically generate and integrate a new meaning between the be connection chain,
- extract a common Interface (pattern).

Acknowledgements

The author expresses his gratitude to Professor Edward Pogossian for supervising the work as well as Sedrak Grigoryan, Sipan Babertsyan and Vahan Margaryan for very valuable discussions.

References

[1] E. Pogossian, "On modeling cognition". *Computer Science and Information Technologies (CSIT11)*, Yerevan, pp 194-198, 2011.

[2] M. Chein and M.-L. Mugnier, *Graph-based Knowledge Representation and Reasoning: Computational Foundations of Conceptual Graphs*. Advanced Information and Knowledge Processing Series, Springer London, 2009.

[3] K. Khachatryan and S. Grigoryan. "Java programs for presentation and acquisition of meanings In SSRGT games", *Proceedings of SEUA Annual conference*, Yerevan, 7p, 2013.

[4] M. A. Ghodrat, "Expression equivalence checking using interval analysis", *Very Large Scale Integration (VLSI) Systems, IEEE Transactions*, vol. 14, no. 8, pp. 830-842, 2006.

[5] K. Khachatryan and S. Grigoryan. "Java programs for matching situations to the meanings of SSRGT games", *Proceedings of SEUA Annual conference*, Yerevan, 5p, 2013.

[6] Искусственный интеллект. - В 3-х кн. Кн. 2. Модели и методы: Спровочник/Под ред. Д. А. Поспелова - М. *Радио и связ*, 1990.

[7] E. Pogossian, "Modeling of meaning processing and its applications in competition and combating games", *Proceedings of SEUA Annual conference*, Yerevan 12p, 2013.

[8] N. Dershowitz, *Rewrite Systems*, Handbook of Theoretical Computer Science, Elsevier Science Publishers, 1990.

[9] P. J. Downey, R. Sethi, and R. E. Tarjan, "Variations on the common subexpression problem", *Journal of the ACM*, vol. 27 no. 4, pp. 758–771, 1980.

[10] R. E. Moore, *Interval analysis*, Prentice-Hall, Englewood Cliffs, N. J., 1966.

[11] M. Chein and M.-L. Mugnier, “Conceptual graphs: Fundamental notions”, *Revue d’Intelligence Artificielle*, vol. 6 no. 4, pp 365–406, 1992.
[12] J. Sowa and E. C.Way, “Inplementing a semantic interpreter using conceptual graphs”, *IBM Journal of Research and Development*, vol. 30, no. 1, pp 57–69, 1986.
[13] Ю. И. Журавлев, *Об алгевраическом подходе к решению задач распознавания и классификации*. Проблемы кибернетики. Наука, Вып. 33. С. 5-68, 1978.
[14] Ю. И. Журавлев, “Корректные алгебры над множествами некорректных (эвристических) алгоритмов”, *Кибернетика,* N. 2, с. 35-43, 1978.
[15] Д. А. Поспелов, *Ситуационное управление: теория и практика*, *Наука*, 1986.
[16] М. А. Айзерман, Э. М. Браверман, Л. И. Розоноер, *Метод потенциальных функций в теории обучения машин*, *Наука*, 1970.
[17] Р. Харалик, “Структурное распознавание образов: гомоморфизмы и размещения”, *Кибернетический сборник*, Пер. с англ. N 19, с. 170-199, 1982.

Submitted 25.12.2012, accepted 20.02.2013.

Իմաստների ընդհանրացման համակցական մոտեցում

Կ. Խաչատրյան

Ամփոփում

Նոր իմաստները ձեռք են բերվում հանրությունից հարցման, փորձի միջոցով բացահայտման և առկա իմաստների համակցման միջոցով: Մենք կատարելագործում ենք իմաստները՝ ներկայացնելով նրանց աբստրակտ դասերի միջոցով, որոնք կապված են միմյանց հետ հարաբերությունների լինել-, ունենալ-, անել- (be-, have-, do) կատեգորիաներով [1, 3, 7]: Տրված իմաստների ընդհանրացման միջոցով հոդվածում ներկայացնում ենք իմաստների կառուցման նոր համակցված ալգորիթմ:

Комбинационный подход к обобщению смыслов

К. Хачатрян

Аннотация

Мы получаем новые смыслы за счет приобретения из общин, откровения от опыта и создания новых смыслов путем объединения имеющихся смыслов. Мы уточняем смыслы с помощью абстрактных классов объединенных быть-, иметь-, делать- категориями отношений [1, 3, 7]. В статье представляется новый комбинационный алгоритм для построения нового смысла обобщения данного набора смыслов.