An algorithmic definition of CG operations based on a bootstrap step

Adil Kabbaj[1], Bernard Moulin[2]

[1] INSEA, Rabat, Maroc, B.P. 6217
akabbaj@insea.ac.ma

[2] Laval university, Computer Science Department
Québec, Québec, Canada G1K 7P4
Bernard.moulin@ift.ulaval.ca

Abstract. Complexity of morphism-based CG operations (maximal join, unification, subsumption, generalization) has been identified and studied by several authors. As a consequence, compromise have been proposed, either by assuming a restricted form of CG or by defining a simple or inefficient CG operation. Independently from this approach, this paper proposes to start a CG operation (like maximal join, unify and projection) by a *bootstrap step*. The bootstrap step is supplied by any information that specifies which concepts in the first CG should be mapped to corresponding concepts in the second CG. Three sources of information for the bootstrap are identified in this paper : a) entry concepts that are given to a CG operation, b) concepts with individual or set as referent and c) co-references that are used in compound CGs. Using the bootstrap step, CG operations are defined and implemented in Prolog+CG 2.5 on simple and compound CGs with sets and co-references.

1 Introduction

Complexity of morphism-based operations on general CGs has been identified and studied by several authors. As a consequence, compromise have been proposed, either by assuming a restricted form of CG or by defining simple or inefficient CG operations [20, 6, 8, 11, 2, 23, 24, 18, 15, 1, 5, and many others]. Independently from this approach, we propose to start a CG operation (like maximal join, unify and projection) by a *bootstrap step*. The bootstrap step is supplied by any information that specifies which concepts in the first CG should be mapped to corresponding concepts in the second CG. We propose three sources of information for the bootstrap:

a) *Start by matching entry concepts* which could be given as parameters to the CG operation. This later should process the two given CGs by first matching its entry concepts. Entry concepts can be provided by a complex operation (such as a type definition expansion which calls a maximal join operation) or by a task (such as a semantic analyzer [21, 14] which is based on the maximal join too). Of course, each call to a CG operation may involve different entry concepts. Entry concepts are related to a specific call to an operation, while some authors suggested to associate a “head/root” to a CG [19, 4, 1], the two are quite different even if entry concepts could play the role of head concepts, without imposing, however, restrictions on CG expressiveness.

b) *Start by matching the concepts with specific referents :* to match a CG g1 with a CG g2, start by matching any concept c1 in CG g1 with the concept c2 in CG g2 so that c1 and c2 have the same specific referent (an individual or a set).

c) *Start by matching co-referenced concepts with partially matched co-references.* Operations on compound CGs could benefit from the presence of co-references which are considered in this paper not only as special relations to deal with, but also as sources for automatic identification of entry concepts. Indeed, matching two compound CGs is a recursive operation [8] (see also [7] for the case of the projection) and while entry concepts could be given as parameters to the first call of the operation, they are not given for the recursive calls. However and as shown in this paper, co-references will produce entry concepts for the embedded calls.

As it can be noted in previous ICCS proceedings, operations on compound CG with co-references received little attention compared to operations on simple CGs. Apart from our previous and current work [8, 9, 11, 12, 17], Chein and Mugnier's team [3, 7] developed a platform that supports the projection operation on compound CG with co-references. But their algorithm computes first the set S of all the possible projections from a CG G1 to a CG G2 without co-reference constraints, then the set S is filtered to keep only those projections which respect co-reference constraints. As an improvement perspective, they noted that “projection computing can be improved by an algorithm that considers as soon as possible restrictions induced by coreference links” [7]. In [8, 17], we defined and implemented an algorithm for *maximalJoin*, *generalize* and *projection* operations that is based on such an improvement. Since that time, the algorithm has been continually refined and extended [9, 11, 12].

The present paper reports the main results and the latest developments of our work concerning this topic: it gives an algorithmic definition of the basic CG operations (maximalJoin, unification, generalization and subsumption) on simple and compound CGs with sets and co-referents. The definition is based on the bootstrap step.

The paper also emphasizes the impact of co-references on the processing of CG operations. These operations were implemented and integrated as primitive operations in the CG activation-based programming language Synergy [12] (without the possibility to have sets as referents) and in the CG-based logic programming language Prolog+CG [11, 13, 14]. Prolog+CG also provides other CG operations such as concOfCG(Concept, CG) and branchOfCG(Branch, CG) which allow for a powerful treatment of the components of a CG, along with operations on the concept type hierarchy. In [14] we describe some Prolog+CG case studies that use these operations.

The paper is organized as follows. Section 2 defines the CG structure considered in this paper: simple and compound CGs with co-reference and sets as referents. Section 3 defines the bootstrap step and presents an algorithmic definition of CG operations (maximalJoin, unification, generalization and subsumption). Section 4 gives some examples, using Prolog+CG, that illustrate our definition of CG operations. Section 5 briefly concludes the paper.

2 Simple and compound CGs with sets and co-referents

Any use of CGs is based on the underlying *support* : types hierarchy, instances declarations and the related operations. Annex1 gives the information used in the examples presented in the paper. Prolog+CG provides several primitives (10) on the support (subType, superType, maxComSubType, minComSuperType, isInstance, addInstance, etc.)[1].

CG. A conceptual graph (CG) in Prolog+CG is a graph of concept nodes related by conceptual relations. Only binary relations are considered (this constraint, assumed also by several authors, is for simplicity and practical purposes only). Prolog+CG provides flexibility in the use of variables inside CGs. A variable can stand for a whole CG, a whole concept (for instance in [Man]-agnt->X , X is a variable), a relation (for instance in [Man]-R->[Eat] ; R is a variable), a concept type or a concept referent (for instance in [A : B]-agnt->[Eat] ; A and B are variables). Such a flexible use of variables enhances the expressive power of the language with respect to CG manipulation.

Concept. A concept is composed of a type and an optional referent. A concept type can be a variable or an identifier which refers to a type defined in the concept type hierarchy. A concept referent can be a variable, an instance (an identifier or a string), a set of instances, a co-referent (represented by a variable), a multi-referent (which has the form "*Number" and corresponds to ''number designator'' in [16]) or any Prolog+CG data: an elementary data such as an integer, a real, a boolean, an identifier, a string or a composed data like a list, a term, a concept or a CG. In [10, 12, 13], a concept is defined as a structure composed of a type, a referent and a descriptor. A similar definition has been proposed by Chein and Mugnier [3, 7]. In the current version of Prolog+CG and in order to respect the standard CG notation [22], we consider the descriptor as a kind of referent and thus, a concept is composed of only two fields.

Examples of concepts :
[Man], [Man : {John, Carl, Henry}], [Cat : x], [Human : *1], [Integer : 25],
[List : (1, 2, 3)],[Term : papa(x, Hicham)],
[Proposition :
 [Man : Carl]<-agnt-[Think]-obj->[Proposition : [Man: Carl]-attr->[Crazy]]]

3 An algorithmic definition of CG operations based on the bootstrap step

We first discuss functional CGs, backtracking CG operations, CG operations as specialization of a generic matching operation, the strategy to use for matching compound CGs and a general specification of the main CG operations. Then, we present an algorithmic definition of CG operations, based on the bootstrap step.

[1] See the site of Prolog+CG for more detail : www.insea.ac.ma/CGTools/PROLOG+CG.htm

Functional CG. As noted in the introduction, several theoretical studies showed the complexity of operations on general CGs. To provide an efficient implementation of CG operations, restricted forms of CG have been proposed in the literature.

Prolog+CG allows for the formulation of a general CG but CG operations are defined on *functional CGs* : conceptual relations are interpreted as functions (or roles) and thus each concept should be linked to different relations [8, 11]. By allowing sets as concept referents, the restriction to adopt functional interpretation of relations is somewhat relaxed. For instance, our maximalJoin operation can't join the two following CGs ga and gb :

```
  ga :  [Person : Jo]<-agnt-[Speak]-agnt->[Person : Jack]
  gb : [Speak]-agnt->[Person : Karl]
```

but it can join CGs ga' and gb, producing the CG gRes :

```
  ga' : [Speak]-agnt->[Person : {Jack, Jo}]
  gb : [Speak]-agnt->[Person : Karl]
  gRes : [Speak]-agnt->[Person : {Jack, Jo, Karl}]
```

Backtracking CG operations. With the functional interpretation of relations and the bootstrap step, we attempt to minimize as far as possible backtracking and thus to prune the huge search space that occurs if unrestricted CGs are considered. Indeed, a backtracking CG operation is expensive for simple CGs and it is much more expensive for compound CGs (recall that a compound CG is a tree of simple CGs, so we have to match two trees of simple CGs !). Also, a backtracking CG operation produces a set of all possible solutions. The interpretation of such a set is sometimes difficult, especiallu when the operation applies on compound CGs. For these reasons, we minimize the possibility of backtracking, unlike Chein's team [7] who proposes a backtracking projection on compound CGs, and unlike Corbett [4, 5] who avoids the problem.

CG operations are defined as a specialization of a generic matching operation. The algorithm below is based on a generic and object-oriented definition of CG operations, proposed first in [8, 17, 9]. The generic matching operation *matchCG* identifies a mapping (morphism) between two CGs (The specificity of each CG operation is considered by the matchCG operation as needed). Thus, the kind of CG operation to be performed becomes a parameter for *matchCG*, in addition to the two CGs to be matched and optionally the entry concepts.

Fixing the strategy for the match of compound CGs. A compound CG is a tree structure whose nodes represent simple CGs. Thus, *matchCG* operates on CGs called input CFGs, each being a tree of simple CGs. In [8, 17, 9], *matchCG* matches the two trees using a depth-first approach. When the function identifies that two concepts (each belonging to one of the CGs to be matched) must be matched, the match is done immediately. This applies to simple concepts (in which the referent is not a CG) and to complex concepts (in which the referent is a CG) as well. However, with depth-first matching, co-references cannot be fully used in order to generate entry concepts for the embedded CGs.

Considering the levels of embedding of a CG within another CG, a co-reference can be thought of as a special relation that relates a concept at level L with another concept at level L+1. Thus, co-reference matching is a cross-over operation: it begins with the matching of two concepts respectively bcated in the two input CGs at level L and terminates with the matching of two other concepts in two embedded CGs of level L+1. So, it is desirable to terminate the matching of all the concepts and relations in the input CGs of level L (and so, to start matching all the co-references that occur at that level) before considering the matching of the embedded CGs of level L+1. In this way, we increase the probability for co-references matching to provide entry concepts for matching CGs of level L+1.

Thus, matching two compound CGs should be done in a breadth-first fashion: we start by matching simple concepts, relations and doing a partial match of complex concepts (matching concept types only, while referent matching is delayed) of the two input CGs. Then, we return to complex concepts to consider the embedded CGs. This approach is expressed by the following algorithm in which the recursive calls to *matchCG* is the last step, after the match of all the elements of the current CGs.

General specification of the main CG operations :

- ***unify(CG g1, CG g2)* :** checks if the first CG g1 can be unified with the second CG g2. Unify g1 with g2 corresponds to verifying that g1 is a sub-graph of g2 modulo the unification of the corresponding concepts.

- ***subsume(CG g1 [, Concept E1] , CG g2 [, Concept E2] [, CG g3] [, Concept E3])* :** checks if g1 subsumes (is more general than) g2 : verifies that g1 is a sub-graph of g2 modulo the subsumption of the corresponding concepts (concept in g1 is more general than the corresponding concept in g2). If the argument g3 is specified, then *subsume* returns in g3 the image of g1 in g2 (the sub-graph of g2 that is isomorphic to g1). If the arguments E1 and E2 are specified, then they represent two concepts in g1 and g2 respectively and they are considered as entry concepts for the operation : it should start by checking that E1 is more general than E2. If the argument E3 is specified, then it refers to a copy of E2 in g3.

- ***maximalJoin(CG g1 [, Concept E1] , CG g2 [, Concept E2] , CG g3 [, Concept E3])* :** returns in g3 the maximal join of the two CGs g1 and g2.

- ***generalize(CG g1 [, Concept E1] , CG g2 [, Concept E2] , CG g3 [, Concept E3])* :** returns in g3 the common generalization to g1 and g2.

After this preliminary discussion, we can now describe our algorithmic definition of CG operations, based on the bootstrap step which is the first step of the algorithm. Due to space limitation, we only present the main steps of the algorithm. If an operation fails (for instance, if types of two concepts can’t be matched, an exception is triggered and the whole matching fails, except for the backtracking case).

Data and auxiliary operations used by the algorithm *:*

- OperCG = {maximalJoin, generalize, subsume, unify}.

▪ *CG as the main data structure : the internal representation of a CG is a couple* <Concepts, Tuples> ; where "Concepts" stands for a vector that contains the CG' concepts, and "Tuples" stands for a set of tuples (following the proposition of Levinson [16]). For instance, the above CG g4 will be represented as follows :

g4 : < {[Man : Karl], [Eat], [Apple], [Knife]}, {agnt(1, 0), obj(1, 2), obj(1, 3)} > where numbers represent pointers to the concepts in the vector Concepts.

▪ *LstConcsMatched :* a list that records concepts matching. An element of this list has the form <C1, C2, C3> where C1 and C2 are pointers to concepts in the two input CGs G1, G2 and C3 is the pointer to the concept in G3 that results from the match of C1 and C2. The list LstConcsMatched is local to the matchCG operation, i.e. each call has its own list LstConcsMatched.

▪ *LstCorefsMatch :* a list that records co-references matching. An element of this list has the form <crf1, crf2, crf3> where crf1 and crf2 are referents of concepts in the two input CGs G1, G2 and crf3 the referent of a concept in G3 that results from the match of crf1 and crf2.

▪ *type(c) & ref(c) :* they return the type and the referent of concept *c*, respectively.

▪ *inCoref(c1, c2) :* a predicate that returns true if the concept *c1* is in co-reference with the concept *c2*. A co-reference is represented explicitly by a variable or implicitly by an individual referent or a set.

▪ *match(operCG, c1, c2) :* a procedure that matches two concepts *c1* and *c2* according to the specified CG operation *operCG.*

▪ *matched(ga, gb) or matched(c1, c2) :* a predicate that returns true if its arguments have been matched.

```
matchCG(OperCG operCG, CG g1 [, Concept c1] , CG g2 [,
Concept c2]  [, CG g3]  [, Concept c3]) {
do the Bootstrap step; // this step will trigger the
matching of pairs of concepts
if LstConcsMatched is empty  then
Search for a tuple t1 in g1 and t2 in g2 /
matchTuple(t1, t2) succeeds; /* this is a special source
for the bootstrap but unlike the other sources, a
backtrack could occur to this point if the whole
matching fails, and the search will continue for other
tuples to be matched  */
loop
Search for a tuple t1 = <r1, c1a, c1b> in g1 and  t2 =
<r1, c2a, c2b> in g2  /
          t1 and t2 haven't been matched and
          (matched(c1a, c2a) or  matched(c1b, c2b) ) :
matchTuple(operCG, t1, t2);
endLoop;
```

```
if operCG = unify or subsume then
    check that all the tuples in g1 has been matched;
else if operCG = maximalJoin then
          add to g3 all the tuples in g1 and g2 that
          haven't been matched;
     endIf;
loop

Search for a triple p = <c1, c2, c3> in the list
LstConcsMatched / c1 and c2 are complex concepts (with
CG as referents) and ref(c3) isn't determined yet :
matchCG(operCG, ref(c1), ref(c2));
           // recursive calls to matchCG
endLoop
} // end_matchCG

matchTuple(operCG, tuple  <r1, c1a, c1b>,  tuple  <r1,
c2a, c2b>) {
          matchConc(c1a, c2a);
          matchConc(c1b, c2b);
          if (operCG <> unify  and <> subsume/2)  then
            addTuple(<r1, c3a, c3b>, g3) ;
}
matchConc(OperCG operCG, Concept  c1, Concept c2)  {
if matched(c1, c2) then exit ;
   matchType(operCG, type(c1), type(c2), T3);
    matchRef(operCG, ref(c1), ref(c2), Ref3);
          // it ignores the case of CG as referents
   Concept c3 := [T3 : Ref3] ;
   AddElem(<c1, c2, c3>, LstConcsMatched);
   if (operCG <> unify  and <> subsume/2)  then
   addConc(c3, g3);
}
matchType(OperCG operCG, Type T1, Type T2, Type T3)  {
          if (operCG == unify or maximalJoin)  then
              maxComSubType(T1, T2, T3);
          else if (operCG == subsume) then
              isSuperType(T1, T2) must return true;
          else if (operCG == generalize) then
             minComSuperType(T1, T2, T3);
}
matchRef(OperCG operCG, Referent R1, Referent R2,
Referent R3) {
          if (R1 and R2 are two CGs) then exit;
                   // the matching is delayed
          if (R1 and R2 are two free variables) then
                  // R1 and R2 could be co-references
          if (<R1, R2, R> ∈ LstCorefsMatch)  then
             R3 := R;  // a co-reference case
          else R3 := generateVariable();
          and  addElem(<R1, R2, R3>, LstCorefsMatch); }
          else execute <operCG, R1, R2, R3> ;
             // operCG is a variable defined on OperCG
}
```

- **unifRef(R1, R2) :** this operation calls Prolog+CG unification. If the two referents are sets, it checks that R1 ⊂ R2, if Ref is an individual and R2 a set, it checks that R1 ∈ R2.
- **subsumeRef(R1, R2, R3) :** the same treatment of sets as unifRef. For the other cases, it checks that R1 == R2 (if the two are individual) or that R1 is a variable.
- **JoinRef(R1, R2, R3) :** it performs union (and coercion) on sets. For the other cases, if R1 and R2 are individual and R1 =/= R2 then it triggers an exception, else R3 receives R1 or R2 depends on which one is a specific referent.
- **GeneralizeRef(R1, R2, R3) :** it performs intersection (and coercion) on sets. For the other cases, if R1 and R2 are individual and R1 == R2, then R3 := R1 else R3 is generic.

Bootstrap step : identifies which concepts in the two CGs that must be matched. This step is central in our algorithm since the set of matched concepts constrains and guides the matching of the two input CGs, and makes (with the functional CGs restriction) the matching quasi-deterministic.

```
1. Match entry concepts if they are given to the CG operation :
   if Entry points c1 and c2 are given as arguments to the CG operation then
         matchConc(operCG, c1, c2);
   endIf
2. Match concepts with identical individual or set referents :
   for any concept c1 in g1 so that ref(c1) is an individual referent or a set and
      there is a concept c2 in g2 so that ref(c2) is an individual referent or a set and
      ref(c1) == ref(c2)
         do  matchConc(operCG, c1, c2);
   endFor
3. Determine entry concepts by treatment of co-reference: many pairs of entry concepts
could be identified by this way.
     match concepts with partially matched co-references : check that any co-referent
     in g1 according to its lower context ga, has an image in g2.  (see Figure 1)

    for any concept c1 in g1 so that inCoref(ca1, c1) with ca1 a concept in the
         lower context ga and there is a concept c2 in g2 so that inCoref(cb1, c2) with
         cb1 a concept in the lower context gb and matched(ga, gb) and
         matched(ca1, cb1) do
                match the concept c1 with c2 :  matchConc(operCG, c1, c2);
   endFor
```

Note : in Figures 1 and 2, each CG can contain many concepts related by relations, but we focus on co-references only.

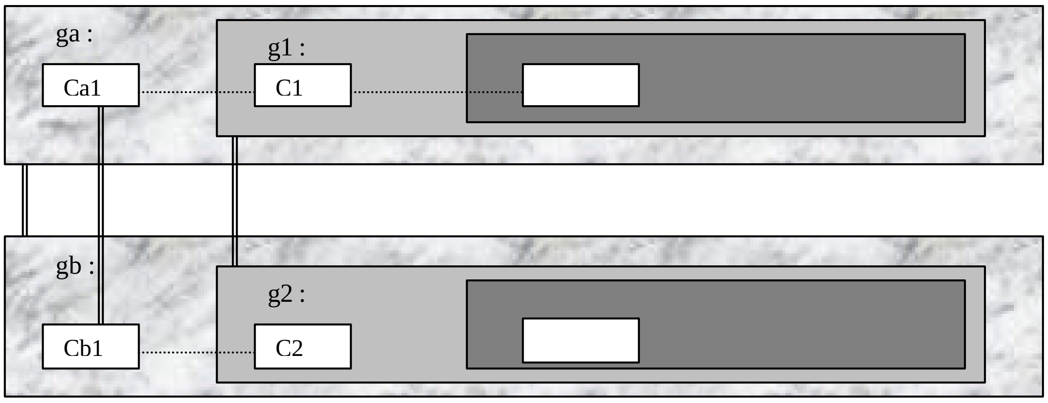

Figure 1: Co-references as a way to determine entry concepts

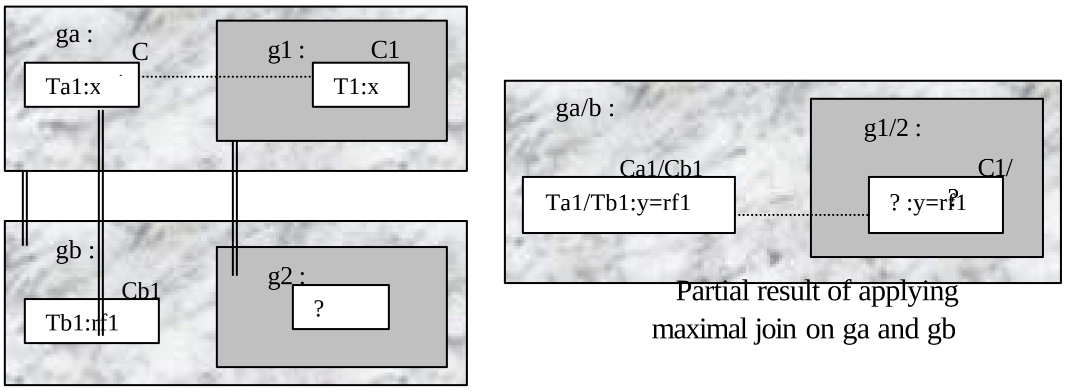

Figure 2: impact of co-reference on maximal join (continue)

There is a relevant special case of co-reference matching which is relevant to the maximal join operation: it should be treated during the operation, not in the bootstrap step (Figure 2). This situation could occur if one of the two CGs contains a co-reference between two, one having a specific referent. In this paper, we illustrate this situation using the following case (Figure 2), the other cases are similar to this one. **If** the two CG ga and gb have been joined (the current level only, without considering the embedded CGs) producing the CG ga/b **and** if the concept Ca1 = [Ta1 : x] in ga has been joined with the concept Cb1 = [Tb1 :rf1] in gb where rf1 represents a specific referent, producing the concept Ca1/Cb1 = [Ta1/Tb1 :y=rf1] **and** if we have to apply a maximal join on g1 and g2 **then** the maximal join operation should check that C1 has no corresponding concept in g2 or that the concept C2 in g2 to be joined with C1 has a generic referent. According to the join of co-reference, it should not have a specific referent different from rf1. In addition, C2 can't have rf1 as a referent, otherwise rf1 corresponds to an individual co-reference and in this case it enters in the general case, treated in point 3 of the bootstrap step).

4 Examples of CG operations with Prolog+CG

This section uses Prolog+CG to present examples of CG operations (maximalJoin, unification, generalization and subsumption). An emphasis is made on the impact of sets and co-references on these CG operations. The type hierarchy and instance declarations used in these examples are given in Annex 1.

a) eq(Concept c1, Concept c2) and eq(CG g1, CG g2) : eq/2 corresponds to unification.

- *Unification on concepts : concepts are considered as Prolog+CG data structures*

```
?- eq([Person : John], [Man : x]).
{x = John}
?- eq([Person : John], [Woman :  x]).
no.
? member([Person : x], ([Man : John], [Color : red],
[Woman : Mary])).
{x = John}
{x = Mary}
```
Unification on simple CGs with sets :
```
it fails since
Andre is not specified in the set {John, Bob, Sam}.
?- eq([Person : Andre]<-agnt-[Drive]-obj->[Vehicle :
myCar],
        [Boy : {John, Bob, Sam}]<-agnt-[Drive]-
          -obj->[Car : x],
          -manr->[Fast]).
no.
?- eq([Person : Bob]<-agnt-[Drive]-obj->[Vehicle :
myCar],
        [Boy : {John, Bob, Sam}]<-agnt-[Drive]-
          -obj->[Car : x],
          -manr->[Fast]).
{x = myCar}
```

- *Unification on compound CGs with co-referents :* the unification succeeds since, apart from the unification of the other components of the two CGs, the co-reference ‘x’ between [Person : x] and [Man : x] in the first CG can be unified with the co-reference ‘Andre’ between [Man : Andre] and [Boy : Andre] in the second CG. Indeed, the concepts [Person : x]/[Man : x] refer to the same entity and this is also the case for the concepts [Man : Andre]/[Boy : Andre].

```
?- eq([Person : x]<-agnt-[Begin]-srce->[Proposition :
[Man : x]<-agnt-[Action]-obj->[Object]],
         [Man : Andre]<-agnt-[Begin]-
          -obj->[Session],
          -srce->[Proposition :
```

```
 [Boy : Andre]<-agnt-[Press]-obj->[Key : enter]-part-
>[Keyboard]]).
{x = Andre}
```

- *Unification on compound CGs with co-referents :* here, the unification fails since the co-reference ‘x’ in the first CG can’t be unified with a corresponding coreference in the second CG.

```
?- eq([Person : x]<-agnt-[Begin]-srce->[Proposition :
[Man : x]<-agnt-[Action]-obj->[Object]],
         [Man : Andre]<-agnt-[Begin]-
           -obj->[Session],
           -srce->[Proposition :
 [Boy : John]<-agnt-[Press]-obj->[Key : enter]-part-
>[Keyboard]]).
no.
```

- *Unification on compound CGs with co-referents :* here, the unification fails again since the co-reference ‘x’ in the first CG has no corresponding co-reference in the second CG.

```
?- eq([Person : x]<-agnt-[Begin]-srce->[Proposition :
[Man : x]<-agnt-[Action]-obj->[Object]],
         [Man]<-agnt-[Begin]-
          -obj->[Session],
          -srce->[Proposition :
 [Boy]<-agnt-[Press]-obj->[Key : enter]-part-
>[Keyboard]]).
no.
```

b) subsume(CG g1 [, Concept E1] , CG g2 [, Concept E2] [, CG g3] [, Concept E3]) :

- *Subsume on simple CGs with sets :*

```
?- subsume([Person]-child->[Person : {John, Sam}],
          [Man : John]-child->[Boy : {Bob, John, Andre,
          Sam}]<-agnt-[Love]-obj->[Girl : Mary]).
{}
```

- *The same request but we specify the third argument to get the image of the first argument :*

```
?- subsume([Person]-child->[Person : {John, Sam}],
          [Man : John]-child->[Boy : {Bob, John, Andre,
          Sam}]<-agnt-[Love]-obj->[Girl : Mary], g).
{g = [Man : John]-child->[Boy : {Bob, John, Andre,
          Sam}]}
```

- *Subsume fails since the set {John, Sam} is not included in the set {Bob, John, Andre} :*

```
?-subsume([Person]-child->[Person : {John, Sam}],
[Man : John]-child->[Boy : {Bob, John, Andre}]<-agnt-
[Love]-obj->[Girl : Mary]). no.
```

- *Subsume on compound CGs without co-references :*

```
?- subsume([Person ]<-agnt-[Begin]-srce->[Proposition :
[Person ]<-agnt-[Action]-obj->[Object]],
[Man]<-agnt-[Begin]-
          -obj->[Session],
          -srce->[Proposition :
[Boy]<-agnt-[Press]-obj->[Key : enter]-part-
>[Keyboard]], g).
{g = [Begin] -
-srce->[Proposition : [Press] -
          -obj->[Key : enter],
          -agnt->[Boy] ],
          -agnt->[Man]}
```

- *Subsume on compound CGs with co-reference :* The same request as the previous one but a co-reference ‘x’ is added in the first argument. *Subsume* fails in this case since the co-reference ‘x’ is not mapped to a co-reference in the second CG : the first CG doesn’t totally subsume the second CG; the information that the two concepts [Person : x] and [Person : x] refers to the same entity is not found in the second CG since the corresponding concepts in the second CG [Man] and [Boy] could refer to different entities.

```
?- subsume([Person :x]<-agnt-[Begin]-srce->[Proposition
:  [Person :x]<-agnt-[Action]-obj->[Object]],
 [Man]<-agnt-[Begin]-
          -obj->[Session],
          -srce->[Proposition :
[Boy]<-agnt-[Press]-obj->[Key : enter]-part-
>[Keyboard]], g).
no.
```

- *Subsume on compound CGs with co-reference :* The same request as the preiou one but a co-reference ‘y’ is added to the second CG. *Subsume* is possible in this case since the constraint imposed by the co-reference in the first CG is verified by a corresponding co-reference in the second CG.

```
?- subsume([Person :x]<-agnt-[Begin]-srce->[Proposition
: [Person :x]<-agnt-[Action]-obj->[Object]],
 [Man : y]<-agnt-[Begin]-
          -obj->[Session],
          -srce->[Proposition :
 [Boy: y]<-agnt-[Press]-obj->[Key :enter]-part-
>[Keyboard]], g).
{x = FREE, y = FREE, g = [Begin] -
          -srce->[Proposition : [Press] -
          -agnt->[Boy : y],
          -obj->[Key : enter]],
          -agnt->[Man : y]}
```

c) maximalJoin(CG g1 [, Concept E1] , CG g2 [, Concept E2] , CG g3 [, Concept E3]) :

- *MaximalJoin on simple CGs with sets :*
```
?- maximalJoin([Person : {Bob, Andre}]<-agnt-[Drive]-
obj->[Car], [Boy : {John, Bob, Sam}]<-agnt-[Drive]-manr-
>[Fast],   g).
{g = [Drive] -
         -agnt->[Boy : {John, Bob, Sam, Andre}],
         -obj->[Car],
         -manr->[Fast]}
```

- *MaximalJoin on simple CGs with coercion and set :*
```
?- maximalJoin([Person : Andre]<-agnt-[Drive]-obj-
>[Car], [Boy :{John, Bob, Sam}]<-agnt-[Drive]-manr-
>[Fast], g).
{g = [Drive] -
         -agnt->[Boy : {John, Bob, Sam, Andre}],
         -obj->[Car],
         -manr->[Fast]}
```

- *Maximal Join failure : coercion impossible since Mary is not conform to Boy :*
```
?- maximalJoin([Person : Mary]<-agnt-[Drive]-obj
            ->[Car], [Boy : {John, Bob, Sam}]<-agnt-
              [Drive]-manr->[Fast], g). no.
```

- *MaximalJoin on compound CGs with co-references :* Since the join of co-reference is considered as a join of a (special) relation, the co-referent 'x' in the first argument is added to the resulting CG g, as a new co-reference c01 that relates [Man : c01] and [Boy : c01] in the CG g ; c01 is a new variable generated by the system. In this example, no special constraint on the maximal join operation has been imposed by treatment of co-reference : the concept [Person : x] in the context [Proposition : [Press] …] which is embedded in the first CG, can be joined with the concept [Boy] or [Person] contained in the context [Proposition : [Boy] …] embedded in the second CG. The first one is choose.
```
?- maximalJoin( [Person:x]<-agnt-[Begin] -
         -obj->[Session],
         -srce->[Proposition : [Press] -
         -obj->[Key : enter]-part->[Keyboard],
         -agnt->[Person : x] ],
         [Man]<-agnt-[Begin]-srce->[Proposition :
           [Boy]<-agnt-[Action]-dest->[Person] ], g).
{x = FREE, g = [Begin] -
         -srce->[Proposition : [Press] -
         -agnt->[Boy : c01],
         -obj->[Key : enter]-part->[Keyboard],
         -dest->[Person]],
         -agnt->[Man : c01],
         -obj->[Session]}
```

- *Impact of co-references on the maximalJoin :* Here, both the first and the second arguments (let's call them g1 and g2) contain co-referents, represented by variable 'x' and 'y' respectively. In this case, treatment of co-references has a great impact on the maximal join : due to the maximal join of the first level of the two CGs, [Person : x] in g1 is joined with [Man : y] in g2 producing [Man : c01] in the resulting CG g, and the context [Proposition : …] in g1 is being joined with the context [Proposition : …] in g2. Now, the maximal join of the contents of these two contexts will be constrained by the join of co-references 'x' and 'y' : the concept [Person : x] in the context embedded in g1 MUST BE joined to the concept [Person : y] in the context embedded in g2. The result is the concept [Person : c02] which is a co-reference to the concept [Man : c02], the two are concepts of the resulting CG g. Note that this graph is different from the resulting CG of the previous example.

```
?- maximalJoin( [Person:x] <-agnt-[Begin] -
      -obj->[Session],
      -srce->[Proposition : [Press] -
      -obj->[Key : enter]-part->[Keyboard],
      -agnt->[Person : x] ],
[Man :y]<-agnt-[Begin]-srce->[Proposition : [Boy]<-agnt-
[Action]-dest->[Person : y] ],   g).
{x = FREE, y = FREE, g = [Begin] -
-srce->[Proposition : [Press] -
      -obj->[Key : enter]-part->[Keyboard],
      -agnt->[Person : c02]<-dest-[Action]-agnt->[Boy]],
      -agnt->[Man : c02],
      -obj->[Session]}
```

d) generalize(CG g1 [, Concept E1] , CG g2 [, Concept E2] , CG g3 [, Concept E3]) :

- *generalize on simple CGs with sets : the case of intersection between sets*

```
?- generalize([Person : {John, Sam, Sue, Mary}]<-agnt-
          [Drive]-obj->[Car]-chrc->[Color : red],
          [Girl : {Sue, Mary, Katy}]<-agnt-[Drive]-
             -obj->[Truck],
             -manr->[Fast],   g).
{g = [Drive] -
          -obj->[Vehicle],
          -agnt->[Person : {Sue, Mary}]}
```

- *Generalize on simple CGs with sets : the case of an element belonging to a set*

```
?- generalize([Person : {John, Sam, Sue, Mary}]<-agnt-
          [Drive]-obj->[Car]-chrc->[Color : red],
          [Girl : Sue]<-agnt-[Drive]-
            -obj->[Truck],
            -manr->[Fast],   g).
{g = [Drive] -
            -obj->[Vehicle],
            -agnt->[Person : Sue]}
```

- *Generalization of compound CGs with co-references :* Generalize processes co-references in the same way as relations : in this example, since the co-reference ‘x’ in the first argument g1 has no corresponding co-reference in the second argument, no co-reference is added to the resulting graph g.

```
?- generalize([Person: x]<-agnt-[Begin] -
           -obj->[Session],
           -srce->[Proposition : [Press] -
           -obj->[Key : enter]-part->[Keyboard],
           -agnt->[Person : x] ],
          [Man]<-agnt-[Begin]-srce->[Proposition :
           [Boy]<-agnt-[Action]-dest->[Person] ], g).
{x = FREE, g = [Begin] -
-srce->[Proposition : [Action]-agnt->[Person]],
-agnt->[Person]}
```

Impact of co-references on generalize : In the current definition and implementation of generalize, we consider the following heuristic about co-reference processing. If both the first and the second arguments (let’s call them g1 and g2) contain co-referents, like in this example (represented by variable ‘x’ and ‘y’ respectively), and if the generalization of the first level of the two CGs involves a common generalization of the two co-references (i.e. generalization of [Person : x] in g1 with [Man : y] in g2 producing [Person : c01] in the resulting CG g), and if the context [Proposition : …] in g1 is being generalized with the context [Proposition : …] in g2, then the generalization of the content of these two contexts will be constrained by the generalization of the co-references ‘x’ and ‘y’.

The concept [Person : x] in the context embedded in g1 MUST BE generalized with the concept [Person : y] in the context embedded in g2. The result is the concept [Person : c01] which is in co-reference with the concept [Person : c01], the two are concepts of the resulting CG g.

Note that this graph is different from the resulting CG of the prevoudexample. As this example shows, with the above heuristic, we preserve the specific information that the two contexts [Proposition …] and [Proposition …] contain the same entity, referring in the first context to [Person : x] and in the second to [Person : y]. However, we loose other information such as the fact that the two contexts contain the information : [Action]-agnt->[Person]. A better solution may be a conjunction of the two information : [Person : c01] and [Action]-agnt->[Person]. Further study is required for the treatment of co-references by the generalization operation (depending on the interpretation given to it).

```
?- generalize([Person: x]<-agnt-[Begin] -
obj->[Session],
srce->[Proposition : [Press] -
-obj->[Key : enter]-part->[Keyboard],
-agnt->[Person : x] ],
[Man: y]<-agnt-[Begin]-srce->[Proposition : [Boy]<-agnt-
[Action]-dest->[Person : y] ],   g).
{x = FREE, y = FREE, g = [Begin] -
```

```
        -srce->[Proposition : [Person : c01]],
        -agnt->[Person : c01]}
```

The simple examples of this section illustrate an important result about the treatment of co-references by CG operations (this result has been identified first in [8]) : co-references can have a great effect on the progress of a CG operation since they impose a specific correspondence between concepts of the two CGs to be matched. In this way, they participate in the reduction of the non-deterministic mapping between concepts.

5 Conclusion

This paper showed that with the bootstrap step and the functional interpretation of conceptual relations, it is possible to provide an efficient algorithmic definition and implementation of the basic CG operations (maximalJoin, unification, generalization and subsumption). The definition applies to simple and to compound CGs with sets and co-references. Moreover, the paper showed the importance and the impact of co-references on the progress of CG operations. Co-references, beside entry concepts and specific referents are the main sources for the bootstrap step.

Annex 1 : The type hierarchy and instances declaration assumed in the paper
In Prolog+CG, a concept type hierarchy is a graph with one root: the type "Universal". It is not mandatory that the type hierarchy be lattice. The hierarchy is formulated by *generalization/specialization rules*, each rule specifies the immediate subtypes of a type. Here is the declaration of the type hierarchy used in this paper :

Universal > Person, Animal, Action, Situation, Object,
AbstractEntity, Attribute.
Person > Man, Woman.
Man > Boy, Employee.
Woman > Girl, Employee.
Employee > Supervisor.
Action > Drive, Love, Break, Rent, Begin, Press, Look.
Object > Vehicle, Machine, Key, Keyboard, Finger.
AbstractEntity > Society, Session, Proposition.
Vehicle > Car, Truck.
Attribute > Fast, Color, Expensive, Big, Age.

We use *instantiation rules* to specify the instances of each concept type :

Boy = John, Bob, Sam, Andre.
Girl = Sue, Mary, Katy.
Color = red.
Key = enter.

References

1. Baader F., R. Molitor and S. Tobies (1999), Tractable and decidable fragments of Conceptual Graphs, In *ICCS'99 proceedings*, LNAI, Springer Verlag. 480-493.

2. Chein M. and M-L Mugnier (1992), Conceptual Graphs : Fundamental notions, in *Revue d'intelligence artificielle,* 6:4. 365-406.
3. Chein M. and M-L Mugnier (1997), Positive nested conceptual graphs, In *ICCS'97 proceedings,* LNAI, Springer Verlag. 95-109.
4. Corbett D. and R. Woodbury (1999), Unification over Constraints in Conceptual Graphs, In *ICCS'99 proceedings,* LNAI, Springer Verlag. 470-479.
5. Corbett D. (2000), A framework for conceptual graph unification, In *ICCS'00 proceedings,* LNAI, Springer Verlag.
6. Fargues J., Landau M-C, Duguord A. and Catach L. (1986), Conceptual Graphs for semantics and knowledge processing, *IBM Journal of Research and Development,* v. 30:1.
7. Genest D. and E. Salvat (1998), A platform allowing typed nested graphs : How CoGITo Became CoGITaNT, In *ICCS'98 proceedings,* LNAI, Springer Verlag. 154-161.
8. Kabbaj A. (1987), *SMGC : un système de manipulation des graphes conceptuels,* M. Sc. Thesis, Université Laval.
9. Kabbaj A., C. Frasson, M. Kaltenbach and J-Y Djamen (1994), A Conceptual and Contextual Object-Oriented Logic Programming : The PROLOG++ language, In *ICCS'94 proceedings,* LNAI, Springer Verlag.
10. Kabbaj A. and C. Frasson (1995), Dynamic CG : Toward a general model of computation, In *ICCS'95 proceedings,* LNAI, Springer Verlag.
11. Kabbaj A. (1996), *Un système multi-paradigme pour la manipulation des connaissances utilisant la théorie des graphes conceptuels,* Ph. D. Thesis, Université de Montreal.
12. Kabbaj A. (1999), Synergy : a conceptual graph activation-based language, In *ICCS'99 proceedings,* LNAI, Springer Verlag. 198-213.
13. Kabbaj A. and M Janta-Polczynski (2000), From PROLOG++ to PROLOG+CG : A CG object-oriented logic programming language, In *ICCS'00 proceedings,* LNAI, Springer Verlag.
14. Kabbaj A., B. Moulin, J. Gancet, D. Nadeau and O. Rouleau (2001), Uses, improvements and extensions of Prolog+CG : case studies, in proceedings of *ICCS'01* (this volume)
15. Kerdiles G. and E. Salvat (1997), A sound and complete proof procedure based on tableaux and projection, In *ICCS'97 proceedings,* LNAI, Springer Verlag.
16. Levinson B. (2000), Symmetry and the Computation of Conceptual Structures, In *ICCS'00 proceedings,* LNAI, Springer Verlag.
17. Moulin B. and A. Kabbaj (1990), SMGC : a tool for conceptual graphs processing, In *The Journal for the integrated study of artificial intelligence, cognitive science and applied epistemology,* 7:1.
18. Mugnier M-L and M. Chein (1996), Représenter des connaissances et raisonner avec des graphes, In *RIA*, 10:1.
19. Müller T. (1997), *Conceptual Graphs as Terms : Prospects for resolution theorem proving,* M. Sc. Thesis, Vrije University.
20. Sowa J. F. (1984), *Conceptual Structures : information processing in mind and machines,* Addison-Wesley.
21. Sowa J. F. and E. Way (1986), Implementing a semantic interpreter using Conceptual Graphs, In *IBM Journal of Research and Development,* v. 30:1.
22. Sowa J. F. (1999), Conceptual Graphs : Draft Proposed American National Standard, In *ICCS'99 proceedings,* LNAI, Springer Verlag.
23. Wermelinger M. (1995), Conceptual Graphs and FOL, In *ICCS'95 proceedings,*LNAI, Springer Verlag. 323-337.
24. Willems M. (1995), Projection and Unification for Conceptual Graphs, In *ICCS'95 proceedings,* LNAI, Springer Verlag. 278-292.