Fachbereich Informatik der Universität Hamburg
Vogt-Kölln-Str. 30, D-22527 Hamburg / Germany

CGPro – a PROLOG Implementation of Conceptual Graphs

Heike Petermann
Lutz Euler

University of Hamburg
Computer Science Department
Natural Language Systems Division
peterman@informatik.uni-hamburg.de
lutz.euler@pink.de

Kalina Bontcheva

Bulgarian Academy of Sciences
Linguistic Modelling Laboratory
kalina@sirma.bg

FBI–HH–M–251 / 95
October 1995

Abstract

Natural language processing requires efficient and powerful tools for representing and processing knowledge. This paper introduces the system CGPro which implements the Conceptual Graphs (CG) formalism. CGs are a logic-based formalism developed by John F. Sowa on the basis of Charles S. Peirce's existential graphs and semantic networks. Conceptual structures proved to be rather convenient as a semantic representation for natural language. CGPro is an efficient and powerful implementation of a Conceptual Graphs representation in Prolog and provides all the operations which are most useful for natural language processing. This paper introduces the functionality of CGPro and describes the motivation for design decisions as well.

Zusammenfassung

Die Verarbeitung natürlicher Sprache erfordert leistungsfähige Werkzeuge zur Repräsentation und Verarbeitung von Wissen. In diesem Papier wird das System CGPro vorgestellt, das den Formalimus der Conceptual Graphs (CGs) implementiert. CGs wurden von John F. Sowa auf der Grundlage der Existenzgraphen von Charles S. Peirce entwickelt. Conceptual Graphs eignen sich besonders gut zur semantischen Repräsentation natürlicher Sprache. CGPro realisiert eine effiziente und mächtige Repräsentation von CGs in Prolog und liefert eine Implementierung der für die maschinelle Sprachverarbeitung wichtigsten Operationen. In diesem Papier wird sowohl die Funktionaltät von CGPro vorgestellt als auch die Motivation der Entwurfsentscheidungen dargelegt.

Contents

1 Introduction

This paper describes in detail the Prolog representation of Conceptual Graphs (CGs) which was used for CGPro and all basic CG operations. With the introduced implementation we aimed at solving the problems associated with representing contexts and n-adic relations of other existing systems.

First, we introduce our representation of graphs, referent fields, individuals and lambda abstractions. Afterwards, we compare it to other existing Prolog representations. Section 3 contains a description of predicates implementing basic access functions for the data structures defined in the previous section. Furthermore we outline the implementation of the "real" CG operations – copy, restrict, join, simplify, maximal join, projection, and a special "extended" projection algorithm.

Special attention was paid to the representation of the referent field. We have chosen feature structures because they can be handled and compared easily. Therefore we are able to obtain the resulting referent field of a join-operation by simple unification of the feature structures.

With CGPro, we satisfy the needs of two application projects – a project for knowledge acquisition from natural language texts and a German-Bulgarian Machine Aided Translation project[2].

[2] "DB-MAT", funded by Volkswagen Foundation for three years (7/1992-6/1995).

2 Representing Conceptual Graphs in PROLOG

2.1 Graph representation

With our representation, we want to fulfill the postulates concerning features of a good CG representation given in (Sowa and Way1986):

- connectivity (traverse the entire graph starting from a concept)
- generality (adequate representation of n-adic relations and several arcs pointing to or from a concept)
- no priviledged nodes (each concept can be a head)
- canonical formation rules (copy, restrict, join, simplify), which are efficiently implemented for the selected representation

All known and available representations (see section 2.8) lack at least one of these features. Therefore, we propose the following representation satisfying all points mentioned above:

```
cg(GraphID, RelationList).
cgc(ConceptID, ContextFlag, ConceptName, ReferentField, AnnotationField).

RelationList    ::=   [cgr(RelName, ArgList, Annotation), ... ]
ArgList         ::=   [ConceptID, ...]
ContextFlag     ::=   normal | special.
ConceptName     ::=   Specialname | Identifier.
Specialname     ::=   context | neg_context | situation | statement | proposition.
```

The basic building blocks in this representation are graphs and concepts. Both are represented as facts, namely **cg/2** ("Conceptual Graph") and **cgc/5** ("Conceptual Graph Concept"). Every graph and concept has a unique identifier (**Id**). Relations between concepts do not have identifiers and occur only as terms in cg/2 facts.

GraphID and **ConceptID** are unique identifiers for all graphs or concepts. Even though the representation is unambiguos concerning the kind of identifiers at each argument place, it is easier to handle only one sort of identifiers for GraphIDs and ConceptIDs.

ArgList is an ordered list of concepts, where the number of the arc corresponds to the concept's place in this list. For an n-adic relation, the arcs are numbered from 1 to n, and the outgoing arc is the last one.

Sowa distinguishes between simple and compound graphs. Simple graphs are those without nested contexts and lines of identity[3]. In this case, the **RelationList** contains the list of the relations of the simple graph, or, if the graph consists of only one concept and thus of no relation, a one-element list with the special relation name **norel**. In the latter case, **ArgList** is a list of only one element.

A compound graph consists of one or more 'toplevel' simple graphs that may contain nested graphs. These toplevel graphs need not be connected directly, but in this case they must contain nested graphs that are connected by a line of identity. In a compound graph the **RelationList** contains the list of all of the relations of the toplevel graphs. It is not distinguished which toplevel graph a relation belongs to. Each of the relations may again be a **norel** relation if the corresponding graph consists of only one concept.

Graph nesting is implemented as follows: A context is established as a special kind of concept whose referent field contains a list of the Ids of the nested graphs.

Coreferent concepts in a graph (i.e. those that share the same variable in the linear form or those that belong to the same line of identity) are represented internally as one concept. As a consequence of this a concept can occur in several contexts at once as long as these contexts belong to the same (compound) graph.

Concepts are represented as 5-Tupels. **ContextFlag** is a flag which is set if the concept is of a special type e.g. context, situation, proposition, and so on. In this case, **ContextFlag** has the value *special*, otherwise it is *normal*. **ConceptName** is a typename or the name of a special context, respectively. For description of the **ReferentField**, see section 2.2 below. Since the possible applications of the **AnnotationField** are not clearly defined, all given examples contain empty annotation fields.

An example of a simple conceptual graph containing a situation is given in figure 1 in graphical notation, in figure 2 in linear form and in figure 3 in the internal representation.

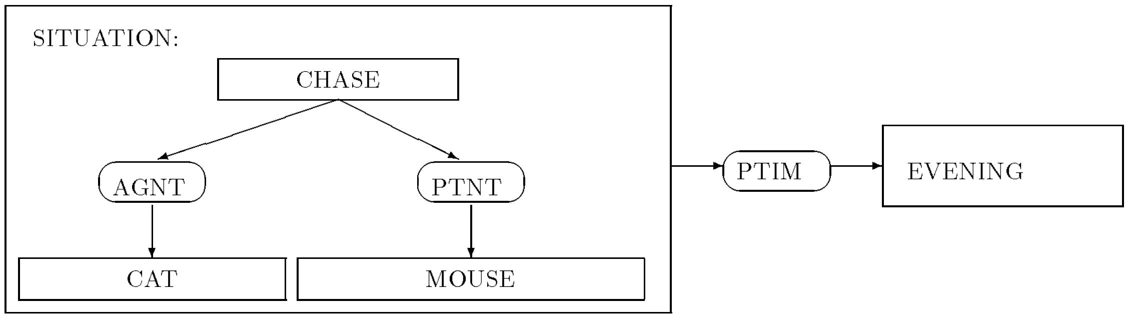

Figure 1: A simple conceptual graph with a situation in graphical form

[3]Note that a line of identity is only a way to show concept equivalence when a concept occurs in multiple contexts and thus is never needed for simple graphs.

```
[SITUATION:
                [CAT] <-(AGNT) <-[CHASE]->(PTNT)->[MOUSE] ]
-> (PTIM) -> [EVENING].
```

Figure 2: The same simple conceptual graph in linear form

```
cg(513, [cgr(ptim, [25, 26], _)]).
cg(514, [cgr(agnt, [27, 28], _), cgr(ptnt, [27, 29], _)]).

cgc(25, special, situation, [514], _).
cgc(26, normal, evening, [fs(num,sing)], _).
cgc(27, normal, chase, [fs(num,sing)], _).
cgc(28, normal, cat, [fs(num,sing)], _).
cgc(29, normal, mouse, [fs(num,sing)], _).
```

Figure 3: The given simple conceptual graph in our representation

2.2 Representation of the Concept's Referent Field

In the basic conceptual graph notation only three kinds of referent fields are permitted – generic (existential), individual marker, and literal. Measures, sets, names, and quantifiers are extended referents, i.e. they are the result of some contraction operation. In our system we want to support all these extended referents, especially the four kinds of set referents – collective, distributive, cumulative and disjunctive (see (Sowa1984, Sowa1992)). Therefore we introduce the following representation for the referent field.

Table 1: Exmaples of the Concept's Referent Field

Referent Field Type	NL translation	Example	Representation
Generic	a book	[BOOK: *]	[num:sing]
Individual	lexicon entry	[PERSON: #123]	[num:sing,type:def,refID:123]
Individual	John	[PERSON:John]	[num:sing,name:'John']
Individual	John	[PERSON:John#1]	[num:sing,type:def,refID:1,name:'John']
Definite ref.	the book	[BOOK: #]	[num:sing, type:def]
Set	John and Mary	[PERSON: {John,Mary}]	[num:plural,type:def, name:['John', 'Mary']]
Partial Set	John,Mary and others	[PERSON: {John,Mary,*}]	[num:plural, name:['John', 'Mary']]
Generic Set	books	[BOOK: {*}]	[num:plural]
Counted Set	three books	[BOOK: {*}@3]	[num:plural, type:meas, quant:3]
Quantifiers	every book	[BOOK: every]	[num:sing, quant:every]
Question	which book	[BOOK: ?]	[num:sing,type:quest]
Plural quest.	which books	[BOOK: {*}?]	[num:plural,type:quest]
Variable		[BOOK: *x]	[quant: lambda]

In table 1 there is a summary of the most frequently used kinds of referent fields together with an example, its natural language reading and the corresponding representation.[4]

```
<Referent>          ::=  ”[” <RefNumber> [”,” <RefType>] [”,” <RefQuant>]
                         [”,” <RefID>] [”,” <RefName>] [”,” <QuantScope>] ”]”

<RefNumber>         ::=  num ”:” <ReferentNumber>
<RefType>           ::=  type ”:” <ReferentType>
<RefQuant>          ::=  quant ”:” <QuantType>
<RefID>             ::=  refID ”:” <IndivID>
<RefName>           ::=  name ”:” <Marker>
<QuantScope>        ::=  scope ”:” <Scope>

<ReferentNumber>    ::=  sing | plural
<ReferentType>      ::=  def | meas | quest
<QuantType>         ::=  every | lambda
<Scope>             ::=  disjunct | dist | col | cum
<Marker>            ::=  <IndName> | <RefSet>
<IndivID>           ::=  Number
<IndName>           ::=  ” ’ ” UpperLetter|SmallLetter ... ” ’ ”
<RefSet>            ::=  ”[” <IndName> [”,” <IndName> ]... ”]”
```

Figure 4: A BNF grammar of the referent field structure

Figure 4 contains a BNF grammar specifying the possible attributes and their values.

The referent field will be represented as a Prolog list of binary terms:

```
[fs(F1,V1), fs(F2,V2), ...  ]
```

Predicates operating on the referent field are described in section 3.3.5.

2.3 Representation of Individuals

Individuals are represented as ternary relations in the knowledge base:

```
ind(IndId,Name,Type)
```

`IndId` must be provided by the user in the referent field of concepts. `Name` is the name of the individual and `Type` is a type of the type hierarchy. Example:

[CAT: TOM #123]	`ind(123, TOM, CAT)`
[CAT: TOM #]	is not an individual as the user has not defined an Id

2.4 Type Definitions

In the conceptual graph formalism new concept types are introduced by type definitions. The operations acting upon them are *type expansion* and *type contraction*. Following Sowa's

[4]Most of the examples are taken from (Sowa1993).

definition (Sowa1984, pp.106 – 112), type definitions are represented as binary relations:

```
typedef(TypeName, lambda(VarList, GraphId))
```

The linear form of type definitions will be interpreted by the linear form parser (see section 4.2.2):

TypeDef ::= "type" <TypeName>"("<VarList>")" "is" < Graph >

`typedef/2` and **Graph** will be asserted in the knowledge base and its GraphId will be included in the `typedef/2` relation. **VarList** is a regular Prolog list of all arguments of the defined type. Example:

type POSITIVE(x) is [NUMBER: *x] -> (">") -> [NUMBER: 0]

will be asserted in the knowledge base in the following relations:

```
cg(1,[cgr("<",[2,3])])

cgc(2,normal,'NUMBER',[quant:lambda],_)
cgc(3,normal,'NUMBER',[type:meas,name:0],_)

typedef('POSITIVE', lambda([x],1))
```

Note that type definitions do not expand the type hierarchy automatically. The user is responsible for providing the corresponding `isa/2` relations.

2.5 Relation Definitions

Relation definitions are supported similarly to type definitions. They will be represented in the knowledge base as binary relations:

```
reldef(RelName, lambda(VarList, GraphId))
```

This relation is a result of parsing the following linear form:

RelDef ::= "relation" <RelName>"("<VarList>")" "is" < Graph >

`reldef/2` and `Graph` are asserted into the knowledge base by the parser.

2.6 Type Hierarchy

The type hierarchy is represented as binary relations:

```
isa(SubType,SuperType)
```

It can be loaded from a file with `loadKBase/2`. `isa/2` relations are not parsed by the linear form parser but are consulted by the Prolog system.

2.7 Attribute Lists

It is very important for the user developing a real world application that there is a possibility for organizing graphs in groups. For this reason, the system supports a mechanism for marking graphs with attributes. Some useful attributes might be: 'canonical', 'typedefs', 'reldefs', 'temp'. For this purpose, attribute lists are provided:

```
attrList(Name,GraphId)
```

In principle the user should take care of managing these lists. The system provides some predicates for this (see section 4.1). All operations changing the knowledge base work with the current attribute list. This variable can be handled with `setCurAttrList/1` and `getCurAttrList/1`.

2.8 Other Prolog Representations

2.8.1 Representing Conceptual Graphs as triples

We would like to introduce the representation adopted in (Hook and Ahmad1992).

Each conceptual graph is decomposed into canonical graphs[5] consisting of two concepts linked by a relation. The investigation made by the authors of (Hook and Ahmad1992) proved that all relations in a Terminology Knowledge Base (TKB) are binary, i.e. connecting exactly two concepts. Their conceptual graph TKB consists of a set of canonical graphs and a type hierarchy. All canonical graphs are stored as:

is_canonical_graph('emission control device' : type : 'catalytic converter').

Each conceptual graph is represented as a Prolog list of triples:

concept : relation : concept

In (Hook and Ahmad1992) the type hierarchy is generalized to a set of conceptual graphs containing the *type* relation. Due to the fact that the inheritance mechanism depends on the type hierarchy, the *type* relation is of particular importance.

This representation handles very simple conceptual graphs. They are suitable mainly for knowledge bases in very specific domains, where all relations are binary. Another problem of that approach is the redundant information. Each concept participating in more than one relation occurs more than once in the list. When specialization is performed, the list should be searched completely for all occurences of a certain concept (the one to be restricted).

[5]Here the term 'canonical' is used in its general meaning. Thus these 'canonical graphs' are not Sowa's 'canonical graphs', but simply a normalized form of a graph representation.

2.8.2 Representing Conceptual Graphs as Concept and Relation lists

(Sowa and Way1986) proposed a more structured representation:

cg(<ConceptList>, <RelationList>)

ConceptList := [cgc(<ConceptNo>, <ConceptName>, <Referent>), ...]

RelationList := [cgr(<ConceptNo>, <RelationName>, <ConceptNo>), ...]

Although one graph is represented as only one data structure, e.g. graph traversing will be easier than in (Hook and Ahmad1992), we do encounter the following problems:

1. Graph referents: [PROPOSITION: Graph] is transformed into

   [PROPOSITION] -> (STMT) -> Graph

   The same holds for STATE, SITUATION and CONTEXT.

2. 3-adic relation representation. Let's take BETW for example. Using the above representation, we are forced to have 2 or 3 entries in the RelationList for one relation. Apart from that, we do not have a clearly defined order of the relation arcs, which contradicts to the fact that the arcs should be numbered for all n-adic relations ($n \geq 3$).

3. There is redundant information in the cgc-structure. If we have the same concepts in different CGs then all that information is included in every cgc-structure. Representing all concepts into a separate concept table avoids this kind of redundancy. Additionally, garbage collection techniques can be applied in order to abandon all concepts not used in the knowledge base.

3 Implementation of some operations on CGs

Although our internal representation handles complex graphs properly, some of the operations are implemented only for simple graphs with situations, propositions, and statements. Our further goal will be to extend our algorithms for complex graphs. But nevertheless the set of operations handled by the system is complete and will remain unchanged.

3.1 Overview

The following sections contain a structured description of our Prolog implementation with predicate names, arguments and brief explanations. First we introduce some abstract data types - *concepts* (see 3.3.1), *conceptual graphs* (see 3.3.2), *types* (see 3.3.3), *individuals* (see 3.3.4) and *referents* (see 3.3.5). A set of standard operations belongs to all ADTs – *contructors, destructors, accessors, copy* and *equality test.* Additionally, each ADT has some specific operations (eg. *subConcept, minComSuperType*), and their semantics is taken from the CG theory.[6] Since we rely on the reader's knowledge of conceptual graphs, we have ommited all functionality details of implemented operations.

The ADTs comprise a basis which the four canonical formation rules and some other CG operations are built upon. All CG operations have both destructive and non-destructive versions. Section 3.4 contains the definitions of *copy, restrict, simplify* and *join. Match, projection* and *maximal join* are introduced in section 3.5. Apart from the standard projection algorithm we have implemented an *extended projection* (see 3.5) that has proven to be rather useful for some Natural Language (NL) applications. *Type and relation expansion/contraction operations* (see 3.6) enable the active use of new types and relations.

In order to distinguish between various kinds of graphs, we have introduced *attribute lists* (see 4.1). They are used to group the graphs according to their semantics (eg. canonical, situations, type definitions, etc.). The system always deals with the *current attribute list.*

Finally, we introduce some predicates for handling the Knowledge Base (KB) - *load, load with convert, save* and *restore* (see 4.2.1). For initializing the knowledge base, we implemented a linear notation parser which converts files containing graphs in linear notation into the internal representation (see 4.2.2).

3.2 General Conventions

3.2.1 Programming Conventions

Concerning uppercase vs. lowercase and the use of underscores, predicate and variable names are represented as follows: `predicateName(VarName, ...)`. Underscores are never used

[6]For a short introduction see (Sowa1992).

except of course as identifier for the anonymous variable.

All predicates obey the following rules concerning the order of arguments:

- Relational binary predicates have their arguments in the order that renders identifying the relation as an infix naturally. Example: `isa(V1, V2)` means `V1 isa V2`.
- Data structure accessors have the output argument in the last position.
- Functional predicates have the output argument in the last position. `join(G1,G2,G3)` joins `G1` and `G2` to yield `G3`.
- Destructive operations have the input/output argument in the first position. `join(G1,G2)` joins `G1` with `G2` and returns the modified `G1`.

3.2.2 Format of the Predicate Descriptions

Below, the predicates are described in the following format:

predicate(Argument1, ...)

Description text

Arguments:

Argument1	Type of Argument1
⋮	⋮

In the first line the argument's names are preceded by one of three mode specifiers: `+` means the argument must be instantiated, `-` means it must be free, and `?` means it can be anything. Note that Id arguments of objects that are modified by the operation have mode `+` because the Id is not changed.

3.3 Concepts, Graphs, Types, Individuals and Referents as Abstract Data Types

Generally, the operations on compound objects comprise constructors (that build a new object from its parts), destructors, accessors (that return the parts of a given object), a copy operation and an equality test. The name of the constructor for datatype *object* is `newObject`, of the destructor `deleteObject` and of the accessors `objectSlot`. The copy operation is named `copyObject` and the equality test `equalObject`.

Also generally all objects are referred to internally by their Ids, e.g. when being passed as arguments. This holds for graphs, concepts and individuals. Some operations on these data structures work destructively. This means that the data structure pointed to by the Id changes while the Id of the input and the result is the same. A nondestructive operation returns a new Id for these kinds of objects.

All operations on graphs take care of creating new concepts where necessary to ensure that any concept is contained in one graph at most. For instance the destructive "join" operation – that modifies one of the input graphs – copies the other input graph so that only new concepts are joined to the first graph.

3.3.1 Concepts

Concepts are represented as 5-tupels:

```
cgc(ConceptID, ContextFlag, ConceptName, ReferentField, AnnotationField)
```

newConcept(+Category, +Type, ?Referent, ?Annotation, ?Concept)

Creates a new concept given its category, type, referent and annotation. Returns the Id of the new concept. For special concepts the referent is a graph Id. If `Concept` is ground it is used as the Id of the concept, else a new Id is acquired and returned.

Arguments:

Category	`normal` or `special`
Type	Atom
Referent	Term or list of graph Ids
Annotation	Term or _
Concept	Concept Id

deleteConcept(+Concept)

Deletes the `Concept`. The user is responsible for assuring that there are no references to this concept left.

Arguments:

Concept	Concept Id

conceptSlots(+Concept, -Category, -Type, -Referent, -Annotation)

Gets the parts of `Concept`.

Arguments:

Concept	Concept Id
Category	`normal` or `special`
Type	Atom
Referent	Term or list of graph Ids
Annotation	Term or _

copyConcept(+ConceptIn, -ConceptOut)

Copy a concept. In case this is a special concept, the contents of the referent field are copied recursively. Returns the Id of the copy.

Arguments:

ConceptIn	Concept Id
ConceptOut	Concept Id

equalConcept(+Concept1, +Concept2)

Succeeds if the two concepts are equal. This is the case if their types and referents are equal. In case of special concepts the nested graphs are compared recursively.

Arguments:

Concept1	Concept Id
Concept2	Concept Id

subConcept(+SubConcept, +SuperConcept)

Succeeds if `SubConcept` is a subconcept of `SuperConcept`.

Arguments:

SubConcept	Concept Id
SuperConcept	Concept Id

conceptCategory(+Concept, -Category)

Get the category of a concept.

Arguments:

Concept	Concept Id
Category	`normal` or `special`

conceptType(+Concept, -Type)

Get the type (from the hierarchy) of a concept.

Arguments:

Concept	Concept Id
Type	Atom

conceptReferent(+Concept, -Referent)

Get the referent field of a concept.

Arguments:

Concept	Concept Id
Referent	Term or list of graph Ids

3.3.2 Graphs

Graphs are represented as binary prolog facts:

```
cg(GraphID, RelationList)
```

newGraph(+Relations, ?Graph)

Create a new graph consisting only of the `Relations`. The new graph is added to the current attribute list. If `Graph` is ground it is used as the Id of the graph, else a new Id is acquired and returned.

Arguments:

Relations	List of terms `cgr/3`
Graph	Graph Id

newGraph(?Graph)

Create a new empty graph. The new graph is put on the current attribute list.

Arguments:

Graph	Graph Id

deleteGraph(+Graph)

Delete the `Graph`. This does not delete the concepts contained in the graph. The user is responsible to assure that there are no more references to the graph. The graph is removed from all attribute lists it is on.

Arguments:

Graph	Graph Id

modifyGraphRelations(+Graph, +NewRelations)

Modify the `Graph` to consist of the relations in the list `NewRelations`.

Arguments:

Graph	Graph Id
Relations	List of terms `cgr/3`

graphRelations(+Graph, -Relations)

Get the list of relations of a graph.

Arguments:

Graph	Graph Id
Relations	List of relations, i.e. terms with functor `cgr`

addGraphRelation(+Graph, +RelName, +Concepts, +Annotation)

Insert n-ary relation with name `RelName` in the `Graph` on the n concepts from the list `Concepts`. The concepts are ordered and the last one belongs to the outgoing arc. Annotate the relation with `Annotation`. The graph is modified.

Arguments:

Graph	Graph Id
RelName	Atom
Concepts	List of Concept Ids
Annotation	Atom

deleteGraphRelation(+Graph, +RelName, +Concepts)

Delete the relation with name `RelName` connecting the `Concepts` from the `Graph`. Fails if there is no such relation.

Arguments:

Graph	GraphId
RelName	Atom
Concepts	List of Concept Ids

copyGraph(+GraphIn, -GraphOut)

Copy a graph.

Arguments:

GraphIn	Graph Id
GraphOut	Graph Id

copyGraph(+GraphOrig, -GraphCopy, -MapOut)

Copy `GraphOrig` and recursively graphs nested in it yielding `GraphCopy`. The mapping of old concepts and graphs onto the new ones is returned in `MapOut`. This can be used to trace concepts and subgraphs through the process of copying.

Arguments:

GraphOrig	Graph Id
GraphCopy	Graph Id
MapOut	Term map/2

The graph copy operation is actually already one of Sowa's CG operations. It is nevertheless defined here because it is needed for completeness of the ADT.

equalGraph(+Graph1, +Graph2)

Succeeds if the two graphs are equal. This is checked recursively for all nested subgraphs, too.

Arguments:

Graph1	Graph Id
Graph2	Graph Id

graphConcept(+Graph, -Concept)

Given `Graph`, succeeds once for each Concept that occurs in the graph. If `Graph` contains nested graphs, the concepts in these graphs are neglected.

Arguments:

Graph	Graph Id
Concept	Concept Id

Automatic Management of Attribute Lists

The `newGraph` operation puts the new graph on the current attribute list. Some operations, e.g. join, use temporary graphs that are deleted in the course of the operation. This is accomplished by using the predicate `deleteGraph` which besides deleting the graph also removes it from all attribute lists. As an additional way to delete graphs there is the predicate `deleteAllGraphs` that deletes all graphs that are on a given attribute list and clears this list. This way the implementation takes care that no attribute list contains a reference to a deleted graph and that every graph is at least on one attribute list.

3.3.3 Types

The type hierarchy is represented as:

```
isa(SubType,SuperType)
```

typelist(-Types)

Returns a list of all types. These include `top`, `bottom` and the special types like `context` etc. The list is topologically sorted in ascending order.

Arguments:

Types	List of types

These operations are undirected binary Prolog relations. 'Undirected' means they can be used both for checking and for generation.

equalType(+Type1, +Type2)

Succeeds if `Type1` and `Type2` are of the same type name.

Arguments:

Type1	Type
Type2	Type

isa(?SubType, ?SuperType)

Succeeds if `SubType` is an immediate subtype of `SuperType`.

Arguments:

SubType	Type
SuperType	Type

subType(?SubType, ?SuperType)

Succeeds if `SubType` is a subtype of `SuperType`.

Arguments:

SubType	Type
SuperType	Type

Ternary relations (directed):

subType(+SubType, +SuperType, -List)

Succeeds if `SubType` is a subtype of `SuperType`. `List` will be unified with the path between `Type1` and `Type2`.

Arguments:

SubType	Type
SuperType	Type
List	List of Types

maxComSubType(+Type1, +Type2, -Type3)

`Type3` is the maximal common subtype of `Type1` and `Type2`.

Arguments:

Type1	Type
Type2	Type
Type3	Type

maxComSubType(+Type1, +Type2, -Type3, -PathList1, -PathList2)

`Type3` is the maximal common subtype of `Type1` and `Type2`. `PathList1` will be unified with the path between `Type1` and `Type3`, and `PathList2` will be unified with the path between `Type2` and `Type3`.

Arguments:

Type1	Type
Type2	Type
Type3	Type
PathList1	List of Types
PathList2	List of Types

minComSuperType(+Type1, +Type2, -Type3)

`Type3` is the minimal common supertype of `Type1` and `Type2`.

Arguments:

Type1	Type
Type2	Type
Type3	Type

minComSuperType(+Type1, +Type2, -Type3, -PathList1, -PathList2, -PathList3)

`Type3` is the minimal common supertype of `Type1` and `Type2`. `PathList1` will be unified with the path between `Type1` and `Type3`, `PathList2` will be unified with the path between `Type2` and `Type3` and `PathList3` will be unified with the path between `Type3` and `univ`.

Arguments:

Type1	Type
Type2	Type
Type3	Type
PathList1	List of Types
PathList2	List of Types
PathList3	List of Types

typeDefinition(+Type, -Definition)

Returns the type definition (a term `lambda/2`) of the `Type`.

Arguments:

Type	Type
Definition	Type definition

3.3.4 Individuals

Individuals are represented as:

```
ind(IndId,Name,Type)
```

The conformity relation must be defined. It has a type and an individual as arguments. The type is the smallest one the individual conforms to. Individuals are referred to by their Id. This is always provided by the user as a number – it is never automatically generated by the system. When parsing a conceptual graph, new individuals occuring inside it are automatically asserted. Additionally, there is a predicate to assert them manually.

newIndividual(+Id, +Type, ?Name)

Create a new Individual. Arguments are the Id, the smallest conforming type and optionally the name. The new individual is asserted.

Arguments:

Id	Number
Type	Type
Name	Atom or anonymous variable

deleteIndividual(+Individual)

Delete the `Individual`. The user is responsible for assuring that there are no references to the individual left.

Arguments:

Individual	Individual Id

individualType(+Individual, -Type)

Get the type of an Individual, i.e. the smallest type the individual conforms to.

Arguments:

Individual	Individual Id
Type	Type

individualName(+Individual, -Name)

Get the name of an Individual.

Arguments:

Individual	Individual Id
Name	Atom or _

3.3.5 Referents

Referents are represented as feature structures (see 2.2).

setRefNumber(+ReferentIn, +Number, -ReferentOut)

Changes the `num`-Feature in `ReferentIn` and returns the result as `ReferentOut`.

Arguments:

ReferentIn	Referent
Number	Atom
ReferentOut	Referent

getRefNumber(+Referent, -Number)

Unifies `Number` with the value of the `num`-Feature in `Referent`.

Arguments:

ReferentIn	Referent
Number	Atom

setRefType(+ReferentIn, +Type, -ReferentOut)

If `type` does not exist as a feature in `ReferentIn`, then `setRefType` adds the feature `type` with value `Type` to `ReferentIn`. Otherwise, `setRefType` changes the value of `type` to `Type` in `ReferentIn`. The result will be unified with `ReferentOut`.

Arguments:

ReferentIn	Referent
Type	Atom
ReferentOut	Referent

getRefType(+Referent, -Type)

Unifies `Type` with the value of the `type`-Feature in `Referent`.

Arguments:

ReferentIn	Referent
Type	Atom

setRefQuant(+ReferentIn, +Quant, -ReferentOut)

If `quant` does not exist as a feature in `ReferentIn`, then `setRefQuant` adds the feature `quant` with value `Quant` to `ReferentIn`. Otherwise, `setRefQuant` changes the value of `quant` to `Quant` in `ReferentIn`. The result will be unified with `ReferentOut`.

Arguments:

ReferentIn	Referent
Quant	Atom
ReferentOut	Referent

getRefQuant(+Referent, -Quant)

Unifies `Quant` with the value of the `quant`-Feature in `Referent`.

Arguments:

ReferentIn	Referent
Quant	Atom

setRefId(+ReferentIn, +Id, -ReferentOut)

If `refID` does not exist as a feature in `ReferentIn`, then `setRefId` adds the feature `refID` with value `Id` to `ReferentIn`. Otherwise, `setRefId` changes the value of `refID` to `Id` in `ReferentIn`. The result will be unified with `ReferentOut`.

Arguments:

ReferentIn	Referent
Id	Atom
ReferentOut	Referent

getRefId(+Referent, -Id)

Unifies `Id` with the value of the `refID`-Feature in `Referent`.

Arguments:

ReferentIn	Referent
Id	Atom

setRefName(+ReferentIn, +Name, -ReferentOut)

If `name` does not exists as a feature in `ReferentIn`, then `setRefName` adds the feature `name` with value `Name` to `ReferentIn`. Otherwise, `setRefName` changes the value of `name` to `Name` in `ReferentIn`. The result will be unified with `ReferentOut`.

Arguments:

ReferentIn	Referent
Name	Atom
ReferentOut	Referent

getRefName(+Referent, -Name)

Unifies `Name` with the value of the `name`-Feature in `Referent`.

Arguments:

ReferentIn	Referent
Name	Atom

setRefScope(+ReferentIn, +Scope, -ReferentOut)

If `scope` does not exists as a feature in `ReferentIn`, then `setRefScope` adds the feature `scope` with value `Scope` to `ReferentIn`. Otherwise, `setRefScope` changes the value of `scope` to `Scope` in `ReferentIn`. The result will be unified with `ReferentOut`.

Arguments:

ReferentIn	Referent
Scope	Atom
ReferentOut	Referent

getRefScope(+Referent, -Scope)

Unifies `Scope` with the value of the `scope`-Feature in `Referent`.

Arguments:

ReferentIn	Referent
Scope	Atom

equalReferent(+Referent1, +Referent2)

Succeeds if `Referent1` and `Referent2` are equal.

Arguments:

Referent1	Referent
Referent2	Referent

matchReferent(+Referent1, +Referent2, -Referent3)

This predicate will be used in join, maximal join and projection. It matches the referent fields of concepts. The following rules are applied in this order:

1. Given that `Referent1` represents a variable then it matches everything if the rest of the referent field is unifyable with the `Referent2`. In this case, `Referent3` is the result of the unification. Otherwise the predicate fails.
2. If rule 1 was not successful and `Referent1` represents a question, then is matches everything if the rest of the referent field is unifyable with `Referent2`. `Referent3` is the result of the unification. Otherwise the predicate fails.
3. If rule 1 and 2 were not successful, `Referent1` is generic or a partial set and `Referent2` is of type measure, then the match succeeds if the number of elements of the set in `Referent1` is less than the number of elements in `Referent2` and the feature structures are unifyable.
4. If none of the rules above were successful, then `Referent3` is the result of feature structure unification of `Referent1` and `Referent2`.
5. Else try to succeed `matchReferent(Referent2, Referent1, Referent3)`.

Arguments:

Referent1	Referent
Referent2	Referent
Referent3	Referent

3.3.6 Miscellaneous

relationDefinition(+Relation, -Definition)

Returns the relation definition (a term `lambda/2`) of the `Relation`.

Arguments:

Relation	Relation
Definition	Relation definition

3.4 The four canonical formation rules

copy(+GraphIn, -GraphOut)

Copy a conceptual graph. In LPA-Prolog this operation is called `copyCG/2` as there is already a system predicate `copy/2` with different semantics. This is the same operation as the 'copy' operation of the graph ADT.

Arguments:

GraphIn	Graph Id
GraphOut	Graph Id

The following operations are provided in two different versions, one that modifies one of the input arguments and one that does not. The destructive versions are useful for avoiding excess garbage generation if several operations are executed for a graph in sequence.

restrict(+Graph, +Concept, +Type, +Referent)
restrict(+GraphIn, +Concept, +Type, +Referent, -GraphOut)

Given a graph and a concept in it, change the type or/and the referent by restrictimg them further. This is allowed only if the conformity relation still holds after the operation. The type may be restricted by replacing it by a subtype. The referent may be restricted by replacing a generic marker by an individual, conforming to the (new) type, or by feature-structure operations, corresponding to specialisation. The first version modifies `Graph`, the second one returns the new `GraphOut` as its result.

Arguments:

Graph	Graph Id
Concept	Concept Id
Type	Type
Referent	Referent
GraphIn	Graph Id
GraphOut	Graph Id

join(+Graph1, +Graph2)
join(+Graph1, +Graph2, -GraphOut)
join(+Graph1, +Concept1, +Graph2, +Concept2)
join(+Graph1, +Concept1, +Graph2, +Concept2, -GraphOut, -ConceptOut)

This joins `Graph1` and `Graph2`. The four versions differ in two aspects, namely whether they modify `Graph1` or not and whether the concepts on which to join the graphs are provided explicitly. In the first two versions the concepts are not provided. They deliver all possible joins on backtracking. In the last two versions `Concept1` is the concept in `Graph1` and `Concept2` is the concept in `Graph2` on which to join the graphs. Both concepts must be equal, otherwise the join will fail. To achieve a join on compatible concepts that are not equal the input concepts must be made equal by restricting them in advance. The resulting concept is provided in `ConceptOut`. The first and third version modify `Graph1` and the second and fourth version provide the result in a new graph `GraphOut`. `Graph2` is always copied in advance and never modified.

Arguments:

Graph1	Graph Id
Graph2	Graph Id
GraphOut	Graph Id
Concept1	Concept Id
Concept2	Concept Id
ConceptOut	Concept Id

simplify(+Graph)
simplify(+GraphIn, -GraphOut)

Simplify a graph by deleting all duplicate relations in it. The first version modifies `Graph`, the second one returns the simplified `GraphIn` as `GraphOut` and leaves `GraphIn` as it was.

Arguments:

Graph	Graph Id
GraphIn	Graph Id
GraphOut	Graph Id

3.5 Match, Projection and Maximal Join

match(+Graph1, Graph2)

Succeeds if `Graph1` is a generalization of `Graph2`. Comparing concepts takes into account the type hierarchy and will match the referents (compare `matchReferent/3`.

Arguments:

Graph1	Graph Id
Graph2	Graph Id

maximalJoin(+Graph1, +Graph2)
maximalJoin(+Graph1, +Graph2, -Graph3)

These predicates realize the maximal join operations for conceptual graphs. For the definition refer to (Sowa1984, p.104). The two versions differ in modifying `Graph2`; `maximalJoin/2` modifies `Graph2` which is the result of this operation. `Graph2` should be more specific than `Graph1`. `maximalJoin` will take into account the type hierarchy and will match the referents (compare `matchReferent/3`).

Arguments:

Graph1	Graph Id
Graph2	Graph Id
Graph3	Graph Id

projection(+Graph1, +Graph2, -Graph3)

Given two conceptual graphs find the projection of `Graph1` on `Graph2`. The resulting `Graph3` is not empty if and only if `Graph2` is a specialization of `Graph1`. The projection operation is type, relation and structure preserving. For the respective algorithm see (Sowa1984, p.99).

Arguments:

Graph1	Graph Id
Graph2	Graph Id
Graph3	Graph Id

extendedProjection(+Graph1, +Graph2, -Graph3)

`Graph3` is the **extended projection**[7] of `Graph2` (the query graph) on `Graph1`. The most important feature is that `Graph2` contains at least one uninstantiated concept (corresponding to a question referent). In a NL application this graph might be created as a semantic representation of a user's question. The resulting graph is obtained by finding the projection

[7] This predicate is not yet implemented.

of Graph2 on Graph1 and adding all concepts directly linked to an uninstantiated concept from the query graph. Let us consider the following graphs:

Graph1: [CAT] -> (ON) -> [MAT] -> (ATTR) -> [RED]

Graph2: [CAT] -> (ON) -> [?]

The result of the normal projection algorithm is just:

[CAT] -> (ON) -> [MAT]

The result of the extended projection algorithm is the whole `Graph1`.

Possible modification of this algorithm includes a parameter for the depth of the extension, i.e. for depth n add all concepts, being n relations 'far' from an uninstantiated concept in the query graph. This parameter can be also dependent on the relation types in the query graph.[8]

Arguments:

Graph1	Graph Id
Graph2	Graph Id
Graph3	Graph Id

3.6 Type and Relation Expansion/Contraction[9]

Since we support both type and relation definitions (i.e. give a mechanism for defining new types and relations), we also provide expansion and contraction operations. In our implementation we stick to the algorithms described in (Sowa1984, pp. 107-109, p.115). It is recommended to apply these operations on canonical graphs, since Sowa has proved that the result is also a canonical graph (i.e. in this case the operations are truth-preserving).

All four operations take as input a graph Id (the graph to be expanded/contracted), an Id of a differentia graph (type definition) or a relator graph (relation definition). The output graph is always the last argument.

[8]For some other ideas about extending the projection operation see (Velardi, Pazienza, and Giovanetti1988).
[9]These predicates are not yet implemented.

typeExpansion(+Graph1, +Graph2)
typeExpansion(+Graph1, +Graph2, -Graph3)

This predicate realizes the type expansion operation on conceptual graphs. `Graph1` is a graph, `Graph2` is the differentia of a type definition and `Graph3` is the resulting graph.
The two versions differ in modifying `Graph1`; `typeExpansion/2` modifies `Graph1` which is the result of this operation.

Arguments:

Graph1	Graph Id
Graph2	Graph Id
Graph3	Graph Id

typeContraction(+Graph1, +Graph2)
typeContraction(+Graph1, +Graph2, -Graph3)

This predicate realizes the type contraction operation. `Graph1` is a graph, `Graph2` is the differentia of a type definition and `Graph3` is the resulting graph from which a subgraph has been deleted and replaced by a single concept (the genus of the type definition).
The two versions differ in modifying `Graph1`; `typeContraction/2` modifies `Graph1` which is the result of this operation.

Arguments:

Graph1	Graph Id
Graph2	Graph Id
Graph3	Graph Id

relationExpansion(+Graph1, +Graph2)
relationExpansion(+Graph1, +Graph2, -Graph3)

This predicate realizes the relation expansion operation on conceptual graphs. `Graph1` is a graph, `Graph2` is the relator of a relation definition. `Graph3` is the resulting graph obtained from `Graph1` after a conceptual relation, and its attached concepts are replaced with `Graph2`.
The two versions differ in modifying `Graph1`; `relationExpansion/2` modifies `Graph1`, which is the result of this operation.

Arguments:

Graph1	Graph Id
Graph2	Graph Id
Graph3	Graph Id

relationContraction(+Graph1, +Graph2)
relationContraction(+Graph1, +Graph2, -Graph3)

`Graph1` is a graph, `Graph2` is a subgraph of `Graph1` and the relator of a relation definition. `Graph3` is the resulting graph obtained from Graph1 after the subgraph `Graph2` is replaced by a single conceptual relation defined by the relation definition.
The two versions differ in modifying `Graph1`; `relationContraction/2` modifies `Graph1`, which is the result of this operation.

Arguments:

Graph1	Graph Id
Graph2	Graph Id
Graph3	Graph Id

4 Service Features

For a more convinient use of the system, we added some service features described in this section. Attribute lists are provided for graph organization. File operations help saving and restoring the knowledge base.

4.1 Attribute Lists

In this chapter, predicates for managing attribute lists are described. The system provides this feature for the user to organize the graphs in groups and to assign attributes to them. Some useful attributes could be `canonical, typedef, temporary`. The user is fully responsible for managing these lists. The system supports a **current attribute list**, and all newly created graphs are inserted there automatically. Therefore the system provides some predicates for managing the current attribute list. All existing attribute lists are saved and restored together with the rest of the KB (see 4.2.1).

getCurAttrList(-Name)

Unifies `Name` with the name of the **current attribute list**.

Arguments:

Name	Atom

setCurAttrList(+Name)

Sets the **current attribute list** to `Name`.

Arguments:

Name	Atom

addAttrList(+Name, +Graph)

Adds `Graph` to the attribute list named `Name`. If there is no such attribute list, it will be created.

Arguments:

Name	Atom
Graph	Graph ID

deleteAttrList(?Name, ?Graph)

Removes the `Graph` from the attribute list named `Name`. The `Graph` itself is not deleted. In case one or both arguments are variables, during backtracking all matching pairs of `Name` and `Graph` are deleted.

Arguments:

Name	Atom
Graph	Graph ID

getAttrList(?Name, ?Graph)

Succeeds if `Name` is the name of an attribute list `Graph` is contained in. If one or both arguments are variables on backtracking all matching pairs are found. This allows finding all graphs on a given attribute list or all attribute lists of a given graph.

Arguments:

Name	Atom
Graph	Graph ID

deleteAllGraphs(+Name)

All graphs that are on the attribute list named `Name` are deleted from the knowledge base, and the list is emptied. `Name` must be completely bound to avoid deleting all graphs on all lists accidentally.

Arguments:

Name	Atom

deleteAllAttrLists(+Graph)

The `Graph` is removed from all attribute lists it is on. It is not deleted itself. `Graph` must be completely bound to avoid accidentally deleting all graphs on all lists.

Arguments:

Graph	Graph ID

4.2 Initializing and Saving the Knowledge Base

4.2.1 Loading, Saving and Restoring the Knowledge Base

loadKBase(+File)

This predicate will load and parse a file, containing the following relations in prolog syntax.

- `cg(CG)` - CG is a conceptual graph satisfying the linear form, including type and relation definitions.
- `isa(Subtype, Supertype)`

The current attribute list is initialized with the value `defaultAttrList`.

Arguments:
 File Atom

loadKBasewithconvert(+File)

This predicate reads a file and converts the content. The user gives the rules for converting `Term1` into `Term2`:

```
convert( +Term1, +Term2 ) :- <converting rules>.
```

It is possible to have more then one converting rule but only one rule per `Term1`. Example:

```
convert(lex(Word,Graph),plex(Word,ID)) :-
        parseCG(ID,Graph).
```

The current attribute list is initialized with the value `defaultAttrList`.

Arguments:
 File Atom

Working with the system it is important to be able to save and restore sessions. Therefore the system offers two predicates. Saving and restoring includes all parts of the knowledge base:

Table 2: Parts of the Knowldege Base to be saved with `saveKBase/1`

conceptual graphs	cg/2 cgc/5
type hierarchy	isa/2
type definitions	typedef/2
relation definitions	relationdef/2
individuals	ind/3
attribute lists	attrList/2
name of current attribut list	curAttrList (global variable)
current identifier	nextId (global variable)

For the exact format of relations in table 2 see section 2.

saveKBase(+File)

Save all parts of the knowledge base in `File`. If `File` does not exist the system will search for file `File.cg`.

Arguments:
File   Atom

restoreKBase(+File)

Restore the conceptual graph knowledge base from `File`. If `File` does not exist the system will search for file `File.cg`. It is important that `File` is a file that has been created by `saveKBase`.

Arguments:
File   Atom

4.2.2 Parsing Conceptual Graphs In Linear Notation

The system provides a parser which converts the linear form to our internal representation (see section 2). The parser accepts conceptual graphs in linear notation as it is specified in (Esch et al.1994).

parseCG(+LGraph, -GraphID)

The predicate `parseCG/2` parses the first argument `LGraph`. The identifier of the parsed graph will be unified with `GraphID` and the graph and all concepts will be asserted into the knowledge base. The graph will be added to the current attribute list (if existing).

Arguments:

LGraph	Atom; CG in linear form
GraphID	Atom

5 Conclusion

This paper has introduced a thorough description of a CG representation in Prolog and the basic graph operations. We have introduced abstract data types (ADTs) for *concepts, graphs, types, referents* and *individuals*. For each ADT, a list of operations has been provided. These data types have been used as building blocks for the implementation of all basic CG operations (e.g. *copy, maximal join,* etc.).

A running implementation exists both for SNI and LPA Prolog. There are some algorithms that need further refinement, but the main part is completed. Additional work aims at the development of new algorithms for proper handling of complex graphs, and the type and relation expansion/contraction operations.

Some further steps are for instance the implementation of a generator for the linear form and a query operation for searching through the knowledge base.

Although we still have a lot of ideas concerning the environment of CGPro it is already now a useful tool for comparing and joining graphs as well as dealing with the type hierarchy.

Acknowledgements

We would like to thank Dr. Galja Angelova for implementing some of the algorithms and her useful comments concerning referent fields, projection and type and relation expansion.

Finally we thank Alexandra Klein and Jens-Uwe Möller for their comments on our work. However, any mistakes that still remain are our own.

References

Esch, John, Maurice Pagnucco, Michel Wermelinger, and Heather Pfeiffer. 1994. Linear - linear notation interface. In Gerard Ellis and Robert Levinson, editors, *ICCS'94 Third PEIRCE Workshop: A Conceptual Graph Workbench*, pages 45–52, College Park, MD, USA. University of Maryland.

Hook, S. and K. Ahmad. 1992. Conceptual graphs and term elaboration: Explicating (terminological) knowledge. Translator's Workbench Project ESPRIT II No. 2315 10, University of Surrey, July.

Sowa, John F. 1984. *Conceptual Structures Information Processing in Mind and Machine.* Addison-Wesley Publishing Company.

Sowa, John F. 1992. Conceptual graphs summary. In Timothy E. Nagle, Janice A. Nagle, Laurie L. Gerholz, and Peter W. Eklund, editors, *Conceptual Structures current research and practice*. Ellis Horwood, chapter I 1, pages 3–51.

Sowa, John F. 1993. Relating diagrams to logic. In Guy W. Mineau, Bernard Moulin, and John F. Sowa, editors, *Conceptual Graphs for Knowledge Representation; First International Conference on Conceptual Structures, ICCS'93; Quebec City, Canada, August 4-7, 1993; Proceedings*, pages 1–35. Springer-Verlag, August.

Sowa, John F. and Eileen C. Way. 1986. Implementing a semantic interpreter using conceptual graphs. *IBM Journal of Research and Development*, 30(1):57–69, Jan.

Velardi, Paola, Maria Teresa Pazienza, and Mario De' Giovanetti. 1988. Conceptual graphs for the analysis and generation of sentences. *IBM J. Res. Develop.*, 32(2):251–267, March.