Textunderstanding in LILOG - Sorts and Reference Objects

Claus-Rainer Rollinger
Rudi Studer
Hans Uszkoreit
Ipke Wachsmuth

IBM Deutschland GmbH
Abt. LILOG 3504
Schloßstr. 70
D-7000 Stuttgart 1

1. Introduction

The main objective of the project LILOG (Linguistic and Logic Methods) is to develop concepts and methods for understanding German texts and dialogs. 'Understanding', in this context, refers to the construction of a semantic representation of a piece of text or of a dialog statement, that is a (partial) model of the situation described in the text. This representation is held in a computer memory and is used by the knowledge processing component, e.g., for extracting information to augment a knowledge base, or for answering questions about the text, etc. As a prerequisite, appropriate means for constructing such a model must be available in a permanent knowledge base. These means must be retrieved and applied to the actual situation by appropriate processes.

The basic processing components of a system capable of understanding German natural language include a parser that analyzes a defined fragment of German and a constructor which produces semantic representations that will be interpreted by a knowledge processing component. While the interactive relationship between syntax and semantics has been dealt with in various approaches, many problem areas central to natural language understanding are still to be attacked. These areas include the syntactic analysis and semantic interpretation of temporal constructions and the adequate treatment of spatial relations.

The body of texts chosen as a focal point for LILOG is concerned with the subject area "region" (as described in touring guides) and thus it heavily employs spatial and temporal constructions. Therefore, a prototpye of a natural language understanding system will be developed that is especially oriented towards the syntactic analysis and semantic interpretation of natural language constructs describing and specifying space and time. Furthermore, as the information conveyed by the text is usually spread across sentences, the analysis and interpretation of sentence connectors will be of special importance.

The means needed to represent textual information combine linguistic and logic-based methods some of which are introduced and described in the following sections of the paper. For representing the different types of knowledge needed within a natural language understanding system, a knowledge representation language, L-LILOG, is currently being designed that is based on a many sorted first-order predicate calculus. In that context, structuring principles for organizing knowledge bases are investigated as well. Ultimately, the concepts and methods developed shall also serve as a basis for knowledge acquisition in the area of advanced expert systems.

The paper is organized as follows. Section 2 contains a brief introduction to the basic concepts and the syntax of the Stuttgart Type Unification Formalism (STUF), which is the underlying representation language for both linguistic and extralinguistic objects in our system. Section 3 presents a description of the knowledge representation language L-LILOG. In particular, the concepts of sorts and of reference objects are introduced which constitute the central means to build semantic representations for textual information. Section 4 gives detailed descriptions of how the STUF formalism is used to realize operations on sorts and reference objects that are to be performed in the process of constructing semantic representations and for utilizing this information.

2. The Stuttgart Type Unification Formalism (STUF)

2.1 The Theoretical Concept

In contemporary formal linguistics, a new and extremely fruitful paradigm has evolved that has already lead to noticeable progress in the area of computational linguistics. The cover term for the new approach is 'unification based grammar formalisms' or simply 'unification grammar'. All unification grammar formalisms employ highly complex representations for linguistic objects--such as lexical entries or syntactic constituents--that are composed of feature-value pairs where the values may either be atomic or again such complex representations. These objects may constrain each other in various ways. The propagation and application of constraints is performed by the same mechanism that serves to build new objects through selectively merging components of old ones, i.e., the declaratively defined operation of graph unification.

This strategy was first introduced by (Kay 1983) and successfully employed in the design of grammar development systems such as FUG (Kay 1983) and PATR-II (Shieber et al. 1983) as well as in the theories and implementations of grammatical frameworks such as GPSG (Gazdar et al. 1985) and LFG (Bresnan et al. 1982). Quite independently, the basic concept was developed by (Aït-Kaci 1984) who worked it out in much more detail as the basis for his lattice-theoretic approach to computation.

The representation formalism STUF is a multi-purpose language for the declaration of complex types and for the declarative specification of operations to be performed on these types. Whereas the formalism was first only used for linguistic processing, it is now being implemented as an abstract data type and will be available as such to all components of the system.

STUF builds on its predecessors. However, promising solutions and intuitive notations from several previous systems are combined in this formalism. Other useful constructs and strategies were added. The language is described in (Uszkoreit 1987).

2.2 The Type Hierarchy

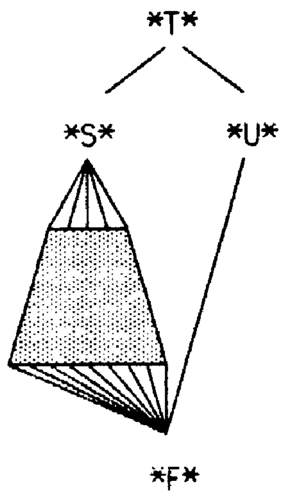
As in previous type unification formalisms, we assume at a certain level of abstraction a lattice of types as the underlying universe of meaningful representations. The expressions of STUF receive their semantics by virtue of the types that satisfy them.

The top element of the type lattice is the empty type *T*. *T* precedes all other types. Since *T* is intensionally empty, its semantic extension is the universe of things U plus the empty set. This may also be expressed in terms of the information contained in a type: Since *T* does not contain any information about its members, it does not constrain membership. The bottom element of the lattice is the failure type *F*, which contains "too much" and therefore inconsistent information. *F* denotes the empty set.

T immediately precedes two other designated types: *U* and *S*. *U* is the type that is assigned to undefined values of functions.[1]
Its denotation is the singleton containing just the empty set. *S* is the top element of the infinite sublattice that contains all other types. *S* denotes the universe of things without the empty set. All types under *S* precede *F*.

The names of the four designated types are the system type names of STUF. All other types may be named by the user. This is the picture of the underlying hierarchy of types:

[1] The purpose of the value *U* cannot be explained here. The type *U* does not only provide a possibility to specify that a type is outside the domain of the function denoted by some attribute. It also permits the definition of fixed-arity-terms as a special case of types with variable arity.



In most type unification formalisms, types are either atomic or complex. Atomic types are simply atoms whereas complex types are partial functions from a finite nonempty set of attributes into the set of types. Atoms may then denote singletons or finite sets of individuals. Every attribute a_1 denotes a function a_1' in $A' \epsilon U^U$. A complex type t_0 with one attribute-value pair $< a_1 , t_1 >$ denotes the subdomain of a_1' for all values in the denotation of t_1, i.e., $[\![t_0]\!] = \{ x \mid \exists y (y \epsilon [\![t_1]\!] \wedge < x, y > \epsilon a_1') \}$.[2]

Complex types may be represented as directed graphs (DG) with labelled edges. Every edge e_1 in a type t is labelled by an attribute a_n and points to the value that a_n has for t. DGs differ from trees in that they can encode equality. Equality is represented as reentrancy; a shared value is pointed at by all attributes that share this value. There is an important difference between the type identity (EQUALity) of values and their token identity (EQuality).

The difference arises through the role that partiality plays in the realm of type unification. All instances of types are inherently partial. Information can be added as long as the type remains consistent. A value shared by token identity cannot be changed without at the same time changing the values of all other attributes that point at the value. The only restriction that is often made to value sharing is the exclusion of cyclicity. According to this restriction, which we adopt for STUF, no node in a graph may strictly precede itself. Equipped with this additional constraint, DGs are also referred to as directed acyclic graphs (DAG).

So far, we have used the term 'precedes' when we referred to the order of our type lattice. However, the relevant weak partial ordering relation is called subsumption, written as $\sqsubseteq$. Subsumption is the reflexive, transitive, antisymmetric closure of immediate subsumption. The corresponding strict partial order is called proper subsumption $\sqsubset$. There is a simple relationship between the relative rank of two types in the subsumption hierarchy and the order of their extensions with respect to each other in the subset lattice:
iff $t_1 \sqsubset t_2$ then $[\![t_1]\!] \supset [\![t_2]\!]$.

The order of the four designated system types with respect to each other is stated in Figure 1. The place of the atomic types in the subsumption hierarchy, i.e., the type lattice, is determined by their extension. If the denotations of all atomic types are unit sets, then every atomic type is immediately subsumed by *S*. In this case, all atoms immediately subsume *F*. If some atomic types denote sets with cardinalities greater than one, they are ordered according to their extensions.

The interesting part is the sublattice of complex types. A complex type t_1 subsumes a complex type t_2 just in case the domain of t_1 is a subset of the domain of t_2 and for every attribute a_i in dom(t_1), $t_1(a_i)$ subsumes $t_2(a_i)$. This condition assures that a type t_1 can subsume a type t_2 if and only if the extension

[2] Actually, the setup in STUF differs a little from the standard definition given here. In STUF there exists no significant difference between atomic and complex types. For the sake of clarity, we shall restrict ourselves here to the simplified story and refer the interested reader to (Uszkoreit 1987).

of t_2 is a subset of the extension of t_1.

The result of unifying two types t_1 and t_2 is the highest type in the subsumption hierarchy that is subsumed by both t_1 and t_2.[3]
If the information contained in t_1 and t_2 is inconsistent, the result of their unification is *F*. Unification on two types corresponds to applying set intersection to their extensions. The generalization of t_1 and t_2 is the lowest type in the lattice that subsumes both t_1 and t_2. Type application as it is proposed in [ref] applies one complex type as a function to another type. For the motivation, definition, and examples of this operation, please refer to (Uszkoreit 1987).

2.3 The Language

The language of STUF has two layers: a powerful notation for the declaration and specification of types and intuitive macro notations for special cases of types that are often needed for certain applications. A description of the macro notations is beyond the aim and the scope of this paper. We will restrict ourselves here to a very brief introduction to the core constructions of the formalism.

The core of STUF is the language for type declaration. User-defined type names are assigned to the specification of a type. The smallest building blocks of a type specification are names of atomic or complex types or minimal types consisting of one attribute-value pair:

car := automobile

mycar := [color: red]

If type names are used to the left of the declaration symbol ":=", they refer to the named type itself. If they occur on the right side of the declaration symbol, they contribute a copy of the contents of the named type to the contents of the type to be declared. In the first example above, a copy of the type 'automobile' is unified with the contents of the type 'car'. If 'car' is declared for the first time, the contents of 'automobile' is unified with the type *T*. Inheritance can be expressed by using the supertypes on the right-hand-side of the declaration of a subtype. The supertypes are not changed since they are called on the right-hand-side.

In place of an attribute, one may specify a path from the node to the value that contains more than one attribute:

somecar := <driver address city> : boston

This definition is equivalent to:

somecar := [driver: [address: [city: boston]]]

There may also be several paths given for one value that are separated by "=". This is one way of encoding value sharing:

person := [<name last> = <father name last> : string]

Type specifications may be composed into new specifications through the operations of unification, disjunction, and type application. Unification is indicated by the empty operator symbol:

[child male]

[color: red make: ford type: sedan]

[3] Since unification is often defined on the inverse of our type hierarchy, this notion corresponds to the one of least upper bound (LUB) in other systems.

Disjunction is indicated by the operator "|":

[make: [ford|toyota]]

The standard notation for functional application was adopted for type application:

[first(alist)]

Values within a type specification may be also be named:

person: = [human [name: [first: string middle: string last: <u>family</u>: = string]]]

These names are only locally scoped, their meaning assignment disappears outside of the type declaration. Local type names may be used as an alternative way to state value sharing. The following two declarations mean the same:

person: = [< name last > = < father name last > : string]

person: = [name: [last: <u>family</u>: = string] father: [name: [last: <u>family</u>: = string]]]

A syntactic calculus provides the equivalences needed for the reduction of the type specifications to canonical forms that can be easily interpreted through their correspondence to sets of types in the subsumption lattice.

<u>3. The Knowledge Representation Language L-LILOG</u>

The knowledge representation language L-LILOG described subsequently has been designed to fulfill two general requirements:

- The language should provide a framework for representing different kinds of domain knowledge which have to be handled in a natural language understanding system. I.e. concepts have to be offered for representing domain specific as well as domain independent real world knowledge which is typically incomplete, vague and/or uncertain. In addition, according to specific topics investigated in the different linguistics subprojects, the representation of temporal and spatial knowledge is of importance as well.
- The language should be based on a sound theoretical basis which provides means (i) for formally specifying the semantics of the language constructs as well as (ii) for formally defining the semantics of the inference processes.

Further, general design requirements for knowledge representation languages (Steels 1984) and several knowledge representation formalisms, especially KRYPTON (Brachman et al. 1983), SRL (Habel 1986), Conceptual Graphs (Sowa 1984), and Discourse Representation Theory (Guenthner et al. 1986), were screened and analysed. Based on this analysis it was decided to design the core of L-LILOG according to the following principles:

1. L-LILOG is based on a many-sorted predicate calculus.
2. Sorts in L-LILOG are partially order-sorted.
3. Knowledge packets offer means for structuring knowledge bases in a hierarchical way.
4. Within the sort hierarchy an inheritance mechanism for attributes is defined.
5. Reference objects are used to represent information available about existing world entities in an object-centered way.
6. Attributes are used for specifying relationships between components of sort specifications.
7. A role-value notation for specifying arguments is offered to support variable arities of predicates.

8. L-LILOG includes different types of notational formalisms among which well-defined transformation rules are given.

Subsequently, we describe some of these principles in more detail. Basically, a L-LILOG knowledge base is defined as follows:

```
Knowledge-Base ::= Sort-Declaration
                   Reference-Object-Declaration
                   Knowledge-Packet-Structure
                   Knowledge-Elements
```

The 'Sort-Declaration' component represents the main part of the LILOG concept lexicon. It contains a specification for each concept known to the system. This information is used to define assertions which are included in the 'Knowledge-Elements' component.

The 'Knowledge-Elements' component is comprised by a set of first-order formulas representing facts and rules which express propositions about real world entities. Roughly speaking, the 'Sort-Declaration' component and the 'Knowledge-Elements' component correspond to the T-Box and A-Box components of representations in KRYPTON-like formalisms (Brachman et al. 1983).

The 'Reference-Object-Declaration' component introduces internal identifiers for the real world entities mentioned in a natural language text. Each identifier is associated with a sort defined in the 'Sort-Declaration' component as well as with a set of designations which have been used in the text to refer to a particular entity. The 'Reference-Object-Declaration' component is based on the referential net concepts described in (Habel 1986).

The 'Knowledge-Packet-Structure' defines a hierarchy of knowledge packets (Wachsmuth 1987) organizing conceptual as well as assertional knowledge in a modular way. To this end, all elements defined in the different knowledge base components are associated with one or several knowledge packet(s) to define the context in which these elements are available to the knowledge processing component. Access conditions are specified which rely on dynamically selecting a specific knowledge packet with respect to which certain parts of the knowledge base are given the status of visible and of reachable knowledge.

3.1 The Sort Concept in L-LILOG

In this section, one of the knowledge base components is presented in more detail: the 'Sort-Declaration' component.

It was decided to use a sort concept within the predicate calculus approach in order to be able to represent and manipulate taxonomic knowledge more appropriately, e.g., by replacing general deduction processes by more specialized processes like type checking (Aït-Kaci 1984). From this, the problem arose how to integrate sorts into the logic based approach. Since sort specifications establish one of the interfaces between the linguistic components and the knowledge manipulation components in the system it was decided to use the STUF formalism for specifying sorts in L-LILOG. As a consequence, a totally homogeneous representation has been achieved which integrates linguistic knowledge and parts of the real world knowledge.

Sort declarations are specified as follows:

```
Sort-Declaration = Sort-set

Sort ::= "SORTDEF" sort-name
         "supersorts:" [sort-name+]
         knowledge-packet-assignment
         [sort-specification]

sort-specification ::= "(" ["properties=" attribute+]
                           ["components=" attribute+]
                           ["elements="  attribute+] ")"
```

```
attribute ::= attribute-name ":"
          [coreference-marker ":"]
          domain-specification

domain-specification ::= sort-name | attribute
```

The 'Sort-Declaration' component consists of a set of sort descriptions each specifying the name of the sort ('sort-name'), its supersort(s), the associated knowledge packets (possibly several), and the sort specification. The partial ordering of sorts has to be a semi-lattice and is intepreted as a set-subset relationship in the models of L-LILOG.

Within the sort specification we distinguish between property attributes specifying properties associated with a sort, component attributes introducing a part-of relationship, and element attributes defining a is-member-of relationship. That is, the three well-known structuring principles for taxonomic knowledge: generalization, aggregation, and grouping are included.

Attributes are specified by their name, optional coreference markers for defining an equality relation between attributes, and a domain specification. Actually, an attribute is interpreted as a partial function from the sort for which it is defined to the sort specified by its domain specification. Sort specifications may be nested to any level.

3.2 The Concept of Reference Objects

Reference objects are introduced as unique internal representatives for external objects of a domain to account for the fact that natural language provides various means to designate a given real world object. The current version of L-LILOG accounts for multiple designations of objects by proper nouns. So far, it does not give consideration to any other definite or any indefinite descriptions of objects. The syntax is given below.

```
reference-object-declaration ::= "REFERENCE OBJECTS"
                                 reference-object+

reference-object ::= reference-object-id "e" sort-name
                 "DESIGNATED-BY <" [designation+] ">"

reference-object-id ::= "r"digit-string

designation ::= object-name
            [knowledge-packet-assignment]
```

These definitions introduce two relations defined between reference objects and other constructs of L-LILOG. The first one is the element relation "e" between reference objects and sorts. By this relation we can specify the sort that a reference object belongs to. The second relation is the "DESIGNATED-BY" relation between reference objects and object names. Object names are proper nouns used in the external world to designate the real object. Instances of this relation enables the system to find out which internal object is referred to when a certain proper noun is used in a discourse to designate an external object. Object names are also needed when we want to generate a natural language sentence that speaks about a real world object.

4. Realization of Operations on Sorts and Reference Objects

4.1 Major Tasks to be Attacked

Having introduced the STUF formalism and the syntax of the part of L-LILOG that pertains to sorts and reference objects we now want to point out which operations we intend to realize with respect to these constructs. At first let us discuss the role that is played by this kind of knowledge in discourse understanding systems.

There are two main components that use knowledge about objects: (1) the linguistic component analyzing natural language sentences in order to construct expressions in the semantic representation language which reflect the meaning of the sentences, and (2) the knowledge processing component that manages the world model of the system, e.g., to answer questions or to modify the model if new information is received.

One major task for the linguistic component is to resolve anaphoric references. That means, it will want to know from the knowledge processing component whether or not two reference objects can be the same from a world knowledge point of view. Another important task is to determine the potential sort(s) of a given object with one or more known attributes, in the process of constructing a semantic representation. A more knowledge-oriented task is the deduction of implicitly given attributes for a reference object.

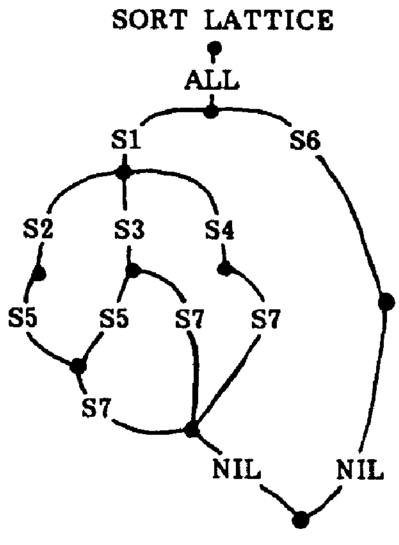
In the remainder of this section we describe the internal representation of the object oriented constructs of L-LILOG as STUF types and discuss the operations for modifying these internal representations and for extracting information out of them. For each particular piece of knowledge to be expressed in L-LILOG with its particular syntactic constructions we give a special internal representation along with operations suited to process this knowledge. For reasons of efficiency and economy the internal representations are process-oriented.

4.2 Sort Declarations

Sort declarations are comprised by two different kinds of information: (1) information about the position of a sort in the sort lattice and (2) attributes that characterize a sort beyond the attributes inherited by this sort from its supersorts. The latter is referred to as 'attribute declaration' hereafter. Attributes are interpreted as one-place functions which map elements of the sort being defined to take values in another sort. The current version does not allow the inheritance of attributes to be blocked so we cannot practice default reasoning at this level.

4.2.1 The Sort Lattice

Sort names correspond to natural language concepts and enable the system to categorize the objects of discourse. For the beginning we assume a fixed lattice of sorts, knowing that this is a simplification. In the long run the sort lattice has to become dynamic to account for the acquisition of new categories. The sort lattice is one special STUF type expressing hierachical relations between sorts. We name this type SORT-LATTICE. Sort names are represented as labels of edges in this type. The supersort relation in the declaration part expresses that a sort Si is the direct supersort of a sort Sj. The path <Si Sj> stands for "Si is a supersort of Sj". If two sorts Si and Sj have a common subsort, then the paths join after Sk: <... Si Sk> = <...Sj Sk> yielding a STUF type like the following:



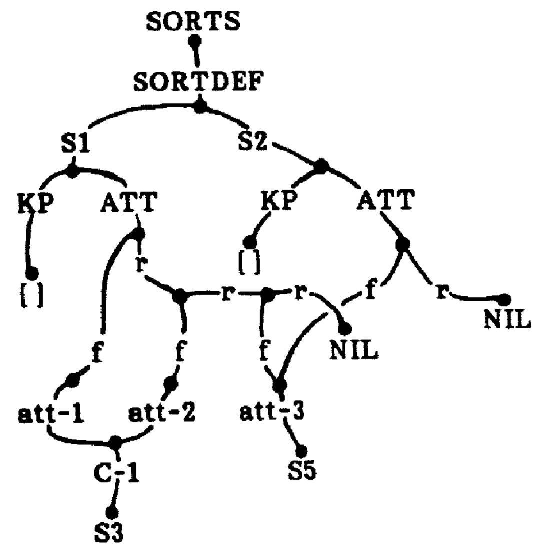
If neither an attribute nor a sort of an object is known, then this object will be assigned to the sort ALL. NIL is the empty sort that completes the lattice. Since the sort lattice is assumed to be static, no operations are defined to change this structure. But we want to exploit information given in the sort lattice to answer the following questions by using STUF operations as indicated:

- Is Si a subsort of Sj?
  (subsort(Si,Sj); STUF: < ALL x* Sj y* Si >)
- Which are the subsorts of Si?
  (subsort(z,Si),fail)
- Which are the supersorts of Si?
  (subsort(Si,z),fail)
- Which is the most general subsort of Si and Sj?
  (Si = Sj) OR (subsort(Si,Sj) THEN Sj) OR (subsort(Sj,Si) THEN Si)
  OR ((most-general-common-subsort(Si,Sj,z);
  STUF: < All x1* Si y1* z > = < All x2* Sj y2* z >))

Here, variables with an asterisk denote functional uncertainty in a search function.

4.2.2 Attribute Declarations

Collecting all attributes of a sort, in particular inherited ones, is the main operation needed for attributes. To this end, the total set of attribute declarations is compiled into a single STUF type, named SORTS, that enables the system to find all attributes of a sort at once, including inherited ones. The attribute of a supersort is shared with each of its subsorts by a coreference link. Then following the path of the supersort allows to find this attribute, following the path of the subsort allows to find the same attribute among others.



In the above example, S2 can be a supersort of S1, because S2 and S1 share the attribute att-3. They can also be sorts at the same hierachical level that both have the same attribute as definition. To avoid conflicts caused by multiple inheritance of the same attribute we require the same value domain to be specified for each occurrence of a particular attribute in a number of sort declarations. As with the sort lattice we assume the type for the attribute declaration to be static. Exploiting information from this type, we want to answer the following questions by using STUF operations as indicated:

- Which are the attributes of a sort and which value domains do they have?
  (get-attributes(Si,x); STUF: type application with the path <SORTDEF Si ATT>)
- Is att-i an attribute of Sj?
  (is-attribute(Sj,att-i); STUF: <SORTDEF Sj ATT r*f att-i>)
- Which is the value domain of an attribute att-i of Sj?
  (get-value-domain(Sj,att-i,x); STUF: <SORTDEF Sj ATT r*f att-i x>)
- Which subsort of Si has an attribute att-j with the value domain Sk?
  (find-subsort(Si,x,att-j,SK):= subsort(x,Si) AND is-attribute(x,att-j),fail)
  If more than one solution is obtained the result is espressed as a disjunction.
- Which most general sorts have the attribute att-i?
  (find-subsort(ALL,x,att-i,y) AND remove all sorts that are subsorts)

4.3 Reference Object Declarations

The declaration of a reference object includes (1) an instantiation of the element relation which relates a particular reference object identifier (e.g. r1) to a particular sort name and (2) an (optional) instantiation of the DESIGNATED-BY relation between r1 and a set of strings (e.g., proper nouns designating this reference object). We handle each case separately.

4.3.1 The Element Relation

For each sort Si we define a STUF type with Si as its name. We take the reference object identifiers of the reference objects that are elements of that sort and associate them as labels with those edges that start at the top node of Si. That way each reference object belongs explicitly to exactly one sort Si and implicitly to all supersorts of Si. This part of our knowledge base is not static: As more information about a reference object is obtained (e.g., that it has a certain attribute) it can be moved down the sort lattice to a subsort. That is, we have to replace "ri ε Sj" by "ri ε Sk", given that "subsort(Sk,Sj)" is true. To do this, <ri> is removed from the type Sj and attached to the type Sk by unification. All reference objects of a sort, including the ones attached to its subsorts can be obtained by unifying type Si with all types of its subsorts.

4.3.2 The DESIGNATED-BY Relation

For each reference object we can define a set of strings designating it. These designations do not have to be definite, that is, one string may designate more than one reference object. For each reference object a type is defined with the reference object identifier as its name. The designations are attached as names to all edges of that type that emanate from the top node.

With respect to the types for the 'element' and the 'DESIGNATED-BY' relation, we can handle the following commands by using STUF operations as indicated:

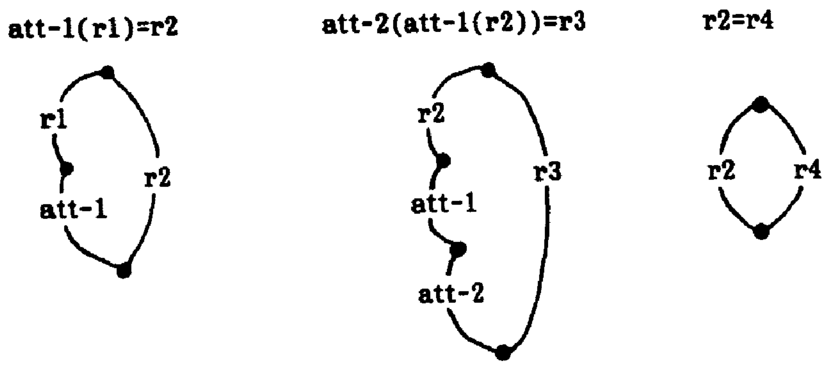
- Which sort does ri belong to?
  (get-refo-sort(ri,x); STUF: x:= < ri >)
- Which reference objects belong to Si and its subsorts?
  (get-all-refos(Si,x):= subsort(x,Si) AND unify(Si,x) for all x)
- What are the designations for reference object ri?
  (get-designations(ri,x); STUF: the type ri)
- Which reference objects are designated by the designation di?
  (get-refo-by-design(di,x); STUF: x:= < di > ; if more than one reference object is designated by di, a set of type names is obtained.)
- Add a designation di to a reference-object ri!
  (add-design(ri,di); STUF: unify(ri, < di >))
- Add a reference object ri to a sort Si!
  (add-refo(ri,Si); STUF: unify(Si, < ri >))
- Change "ri ε Sj" into "ri ε Sk"!
  (change-element(ri,Sj,Sk); STUF: remove(Sj, < ri >) AND unify(Sk, < ri >))

4.4 Instantiated Attributes

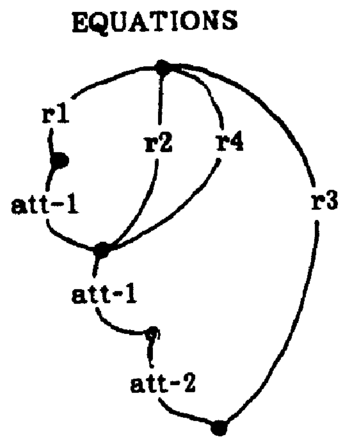
In Section 3.1 attributes were defined as one-place functions. To instantiate the attribute of a reference object means to write expressions of the special kind "term = term" or of the kind "atomic formula". Here, terms can only be simple terms or one-place functions. Specifying the known attributes of the reference objects (either used in the the background knowledge or in the discourse knowledge) means to formulate a special system of equations. This system need not be complete. That means, we do not have to know all attributes of an object, there may be reference objects we only know the sorts they belong to. We are interested in the equality of the attribute values but not in the equality of the attributes as functions. Reference objects of a certain sort may obtain attributes that are not defined for this sort. These additional attributes are required to be attributes of a subsort so we can be sure that the objects with this attribute are elements of that subsort. At the current stage we do not allow to define "free" attributes that are not inherited as we do not allow the blockage of inheritance yet.

The system of equalities is to satisfy the following consistency conditions: (1) a reference object cannot be used as its own attribute value and (2) if a reference object ri is the attribute value of some other reference object rj, then rj cannot be the value of any attribute of ri. In case the second restriction turns out too strong it can be weakened by introducing inverse functions for those functions which are injective. In the internal representation we have chosen for this system of equalities the consistency conditions are easily verified since their violation yields cycles in the resulting type which has to be acyclic by definition.

For each atomic formula we construct as internal representation a type with the two reference object identifiers as labels of the two edges emanating from the top node. These are followed by edges labeled with those attribute names the reference object identifiers are imbedded in. The resulting two paths corefer to the bottom node of the type. Then all such types are unified. The resulting type will contain for each reference object, if at least one of its attributes is known, exactly one edge with its reference object identifier as label. In this way all equations of interest are made explicit. The unification of two types fails if a cycle is produced as defined above.



The unification of these types results in the type EQUATIONS:



Now we can prove equations, e.g., att-2(att-1(att-1(r1))) = r3. To do this, we first construct the corresponding type Gi. Then we test whether or not this type subsumes the equality type. If "subsumes(EQUATIONS,Gi)" is true we have proved that the equation holds. This is the most important operation on the type Gi. Other operations are indicated below:

- Which term is the value of an attribute att-i of an object rj?
  (get(att-i(rj),x); STUF: < rj att-i > = < x* >)
- Which reference object has ri as value of an attribute att-j?
  (get(att-j(x),ri); STUF: < x att-j > = < ri >)
- What are the attributes of the reference object ri?
  (get(x(ri),y); STUF: < ri x* > = < y* >)
- What is the relation between ri and rj?
  (get-rel(ri,rj,z), z is a type; STUF: < ri x* > = < rj y* >)
- Which terms embedded in att-i have which value?
  (get(att-i(x),y); STUF: < x* att-i > = < y* >)
- Add an equation!
  (unify(EQUATIONS,Gi) AND store the resulting type)

4.5 Complex Operations

Now we have the inventory to deal with more complex tasks such as described in Section 4.1. The first one was the question, issued by the linguistic component, if two terms (in the restricted sense used here) can be equal. A more specific question is whether or not two reference objects can be equal. Starting with the second question, we show how to answer them using the internal representations introduced above together with the operations defined on them.

To find out whether or not two reference objects ri and rj can be equal we have to do the following:

- In the first step we test if ri and rj are explicitly known to be equal. We can do this by using the operation "subsumes(EQUATIONS, < ri > = < rj >)". If this test fails we cannot be sure that ri and rj are unequal because we have no closed world assumption.

- In the next step we test whether or not < ri > = < rj > can be unified with the type EQUATIONS. If this test fails, too, then we know that the consistency condition is violated. In this case we have proved that ri is unequal to rj.

- If the unification was successful we further have to check the sorts of the reference objects by using: get-refo-sort(ri,x) AND get-refo-sort(rj,y). x and y must be compatible, otherwise ri and rj cannot be equal. Here, 'compatible' means that

  - x and y are equal: "x = y", or
  - x is a subsort of y or vice versa: "subsort(x,y) OR subsort(y,x)", or
  - x and y have a most general subsort unequal to NIL: "most-general-common-subsort(x,y,z) AND z = / = NIL".

  If one of these operations yields true, we not only know that ri and rj can be equal, we also know the sort they both must belong to if we decide (from outside) that they are equal.

To equate two reference objects requires that they can be equal. If performing the 'can-be' test was successful, the sorts of both reference objects and their ultimate sort is known. The rest to be done is simple. First we unify < ri > = < rj > with EQUATIONS and store the resulting type. Then we have to change the element relation between these reference objects and their sorts if they are unequal. We do not change the DESIGNATED-BY relation but continue to use both reference object identifiers in the factual knowledge base. By doing so we can still differentiate between the designations used in one context from those used in another context.

To answer the more general question, whether or not two terms can be equal, things are more complicated in only one aspect. If a term contains nested functions we compute the sort of this term by computing the embedded terms recursively, starting with the one embedded most deeply, that is, the reference object identifier. The rest to be done is very similar.

5. Conclusion

In our paper, we have tried to sketch the basic design features of our integrated representation system for different kinds of knowledge. A language that is based on a many-sorted predicate calculus has been equipped with the additional descriptive power of a type-unification- based sort hierarchy with equality. The integration of the two systems provides a high degree of conceptual clarity and supports a modular implementation in which the underlying data type and its operations are shared by all components of the system.

6. References

Aït-Kaci, H. (1984). A Lattice Theoretic Approach to Computation Based on a Calculus of Partially Ordered Type Structures. Ph.D. Thesis, University of Pennsylvania.

Brachman, R.J., Fikes, R.E., & Levesque, H.J. (1983). KRYPTON: Integrating terminology and assertion. Proceedings AAAI-83 (pp.31-35).

Bresnan, J. (Ed.) (1982). The Mental Representation of Grammatical Relations. Cambridge, Mass.: MIT Press.

Gazdar, G., Klein, E., Pullum, G., & Sag , I. (1985). Generalized Phrase Structure Grammar. Cambridge, Mass.: Harvard University Press.

Guenthner et al (1986). A theory for the representation of knowledge. In: IBM Journal of Research and Development, Vol.30, No.1, 1986, pp 39-56.

Habel, Ch. (1986). Prinzipien der Referentialität: Untersuchungen zur propositionalen Repräsentation von Wissen. Berlin: Springer.

Kay, M. (1983). Unification Grammar (Technical Report). Palo Alto, Calif.: Xerox Palo Alto Research Center.

Shieber, S., Uszkoreit, H., Pereira, F., Robinson, J., & Tyson, M. (1983). The Formalism and Implementation of PATR-II. In Grosz, B. & Stickel, M. (Eds.) Research on Interactive Acquisition and Use of Knowledge. Menlo Park, Calif.: AI-Center, SRI International.

Sowa, J. (1984). Conceptual Structures: Information Processing in Mind and Machine. Reading, Mass.: Addison-Wesley.

Uszkoreit, H. (1987). STUF: A Description of the Stuttgart Type Unification Formalism (LILOG-Report 16). Stuttgart: IBM Deutschland.

Steels, L. (1984). Design Requirement for Knowledge Representation Systems. In: Proc. GWAI-84, pp. 1-19.

Wachsmuth, I. (1987). On structuring domain-specific knowledge (LILOG-Report 12). Stuttgart: IBM Deutschland.