A reinterpretation component for an NLP system

Ruth Fuchss (fuchss@coli.uni-sb.de)
Sebastian Padó (pado@coli.uni-sb.de)
Universität des Saarlandes, Computerlinguistik

1 Introduction

Cases of reinterpretation present a challenge for semantic processing. Here the result of semantic construction for an expression must be augmented by extralinguistic information in order to avoid an impending conflict, characteristically a sortal or type-theoretic conflict, between constituents of the expression. As an example consider the sentence "Peter is parked out back.". There is a sortal mismatch between NP and VP here, as "Peter" introduces a property of properties of persons, and "is parked" introduces a property of vehicles.

We implemented a reinterpretation component for an existing natural language processing system at Saarbrücken University, the CHORUS system. It is based on Egg's work (Egg 1999; Egg 2000) on this topic, which describes the semantics of reinterpretation cases in terms of semantic underspecification. By introducing new material in the form of *reinterpretation operators*, which mediate between potentially conflicting constituents, we avoid the conflicts. The reinterpretation operators model information not given in the syntax and semantics of the original expression.

In this paper, we describe our reinterpretation component. First, we introduce the formal background of our work, the underspecification formalism we assume for our analyses, and the type-theoretic foundation of our algorithm. Then we give a description of the algorithm.

2 Formal background

2.1 The underspecification formalism

The underspecification formalism we use is the Constraint Language for Lambda Structures (CLLS). CLLS was developed at the Universität des Saarlandes and was originally aimed at modeling scope ambiguities. It is described in detail in (Egg, Niehren, Ruhrberg, and Xu 1998), we will just give a very short introduction here.

CLLS constraints, which can be depicted as graphs, describe lambda-structures. Lambda-structures are augmented tree structures that correspond to lambda.terns (modulo α-equivalence). Lambda-terms that are compatible with the constraints are calles *solutions*.

The constraints comprise the following constructors: Application, denoted by the @-sign, explicit lambda abstraction, denoted by the λ-sign, and lexical nodes. Solid edges represent functor-argument-relations.

Slashed lines depict variable binding. Dotted lines represent dominance edges, which determine the relative position of fixed parts of the representation, so-called fragments. Two examples of CLLS-graphs can be seen in figure 1. Below the graphs we give an intuition of how graphs correspond to lambda terms. This is only an indicative representation. For example, we neglected the upper dominance relation because it is of no interest for our present purposes, as will become clear later.

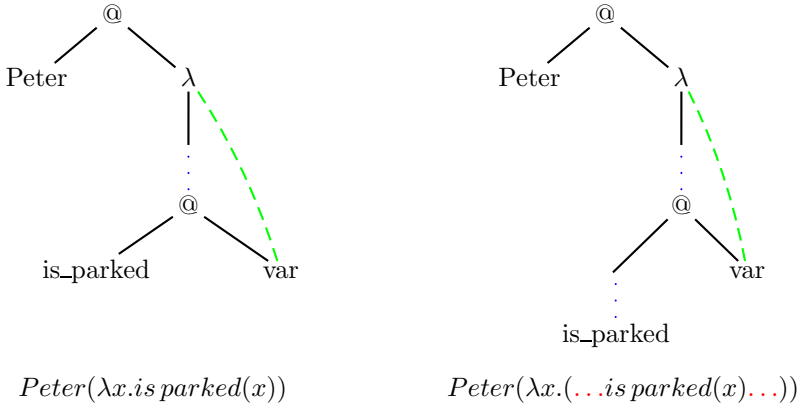
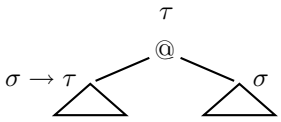


Figure 1: Examples of CLLS graphs

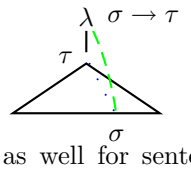
2.2 Type-theoretic foundation

How can one detect sortal conflicts? In type-theoretical semantics, each word is assigned a type. This type information can be augmented by sorts that express *selection restrictions*. In this manner, the semantics of "live" is of type $e_{animate} \rightarrow t$ because living is a property of all animate beings, while the semantics "love" is of type $e_{object} \rightarrow (e_{human} \rightarrow t)$. If there was no more to these types than can be treated in a standard lambda-calculus style, the types of complex semantic expressions would be easy to calculate with the following two rules (let Γ be a type environment). Next to the two rules we show how the lambda-term rules can be transferred to CLLS graphs.

- Application rule: $\dfrac{\Gamma \vdash M : \sigma \rightarrow \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$



- Abstraction rule: $\dfrac{x : \sigma \quad \Gamma \vdash M : \tau}{\Gamma \vdash \lambda x M : \sigma \rightarrow \tau}$



Unfortunately, this treatment would produce sortal conflicts as well for sentences like "Peter lives" because "lives" would semantically be a function $e_{animate} \rightarrow t$ while the semantics of "Peter" would be of type e_{human}, colliding with the required argument type $e_{animate}$. So strict sortal typing cannot suffice. In fact, verbs always accept arguments of *more specific sorts*. To model this, we arrange the sorts in a *sort lattice* which is ordered by the sortal subtype relation (depicted by $\leq$). We show a part of this sort lattice in fig. 2. In our implementation, every entity is assigned exactly one sort.

The formal properties of the sort lattice are captured in the following set of rules that has to be added to the two first rules, allowing more specific arguments than required to be used at any application (let Γ be a type environment and Σ the sort lattice):

- Weakening rule: $\dfrac{\Gamma \vdash M : \tau' \quad \Gamma \vdash \tau' \leq \tau}{\Gamma \vdash M : \tau}$

  This rule states that we can always substitute specific arguments for more general ones. The remaining rules define the properties of the subtype relation.

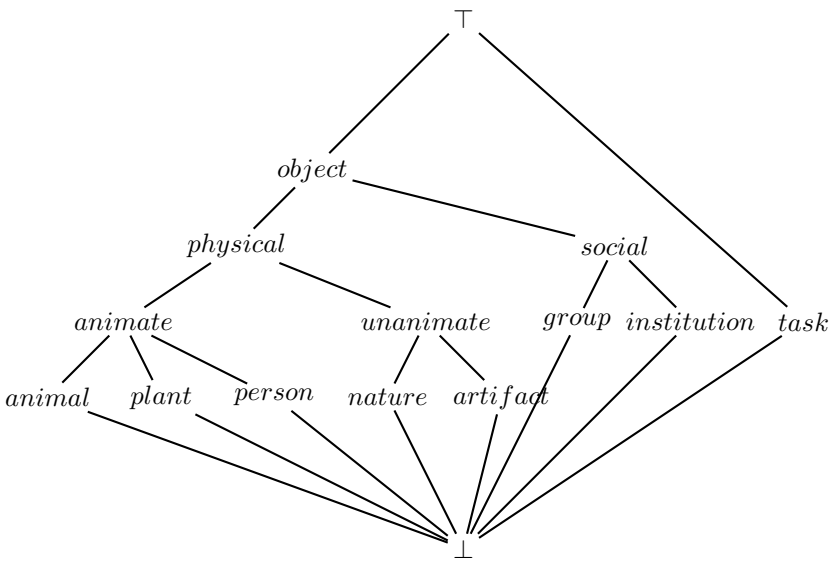


Figure 2: sort lattice

- t-subtyping rule: $\dfrac{}{\Gamma \vdash t \leq t}$
- Sort subtyping rule: $\dfrac{\Sigma \vdash s_1 \leq s_2}{\Gamma \vdash e_{s_1} \leq e_{s_2}}$
- Functional subtyping rule: $\dfrac{\sigma' \leq \sigma \quad \Gamma \vdash \tau \leq \tau'}{\Gamma \vdash \sigma \rightarrow \tau \leq \sigma' \rightarrow \tau'}$

Interesting about the functional subtyping rule is that the argument type of the more specific function is itself less specific. This may not seem intuitive at the first glance, but this formula does indeed provide the correct subtype relations for functions. For example, properties of animate beings ($e_{animate} \rightarrow t$, like "live") have a more specific type than properties of humans ($e_{human} \rightarrow t$, like "talk"), because every property of animate beings is a property of humans but not vice versa.

We come back to the example mentioned in the introduction: let us assume a naive representation of the sentence that assigns type e_{person} to "Peter" and $e_{artifact} \rightarrow t$ to "is parked". We leave out "out back" to simplify the account because it does not contribute anything to the conflict. Using the typing rules, we cannot derive a typing for $is\_parked(Peter)$:

$$\frac{is\_parked : e_{artifact} \rightarrow t \quad Peter : e_{person}}{is\_parked(Peter) :}$$

The type of "Peter" and the argument type of "is parked" are incompatible. Not even the weakening rule can provide a solution, because e_{person} and $e_{artifact}$ do not stand in a subtype relation. This is the kind of problem that can be dealt with in our reinterpretation component.

3 Implementation

3.1 Strategy

On the basis of these theoretical considerations, we now present how we realized the reinterpretation process in the CHORUS system. Our strategy is as follows: During semantic construction, we relax the description in order to create a gap where new material can be added. Then the types are calculated. If a sortal conflict appears, the material that was introduced by semantic construction exhibits impending conflicts which are avoided by inserting a reinterpretation operator.

3.2 Relaxation

We open up gaps at places where reinterpretation can potentially take place by simply inserting a dominance edge. We restrict ourselves to reinterpretations of the verb. This *relaxation* already takes place during the semantic construction. The effects of the relaxation are shown graphically in figure 1. The left hand side shows a CLLS graph without relaxation, while the graph on the right is the relaxed one.

There may be different sources for dominance edges in a constraint. They can arise due to scope ambiguities or due to reinterpretation. E.g. in the right hand graph of figure 1, the upper dominance relation and the lower dominance relation instantiate these two kinds of dominance relation. One might think that there could be unwanted interaction between both kinds of dminance relations. However, (Koller, Niehren, and Striegnitz 2000) shows that, for the examples we treat in our paper, this is not the case. So we can concentrate on the dominance edges we created for reinterpretation.

3.3 Challenges concerning the implementation

We encountered the fundamental problem that only partial information is available. This problem originates from several sources.

Semantic underspecification: No type information can be transported over dominance edges. So we may not be able to determine the types of all nodes completely. Especially isolated fragments that arise e.g. from embedded clauses are critical in this respect. Additional scarcity of information is added by the relaxation, which separates the verb semantics into its own fragment.

Type-theoretic underspecification: The most problematic issue is the sortal underspecification the weakening rule introduces. Applying this rule, there is no longer one canonical solution for the typing of a graph. Rather, we only obtain upper or lower limits for the types and sorts of every node.

To deal with these problems, we introduce the concept of *type constraints*. We collect the type constraints for all nodes that are derived by the typing rules stated in section 2.2. Then we merely *check the existence* of a consistent typing by testing satisfiability of the type constraints.

3.4 Constraints

3.4.1 The constraint paradigm

Constraints are a very general model for processing partial information. Small parts of information are collected incrementally and are allowed to interact. So even incomplete information can be used to its whole extent. For our purposes it is especially interesting that the satisfiability of the information can be tested at any stage.

We use the programming language Mozart Oz because it strongly supports the idea of constraints (see (Schulte, Smolka, and Würtz 1994)). For example, it provides so-called calculation spaces to collect the constraints and check for satisfiability. Thus, type constraints can be directly evaluated.

3.4.2 Type constraints

To obtain the type constraints, we create a *type variable* for every node in our CLLS graph. During semantic construction, we collect the type restrictions for every node in every local tree according to the typing rules introduced in section 2.2 and the entries of the lexical nodes which carry type and sort information. Crucial for the propagation of type constraints are the application and abstraction rule, which we described as lambda term rules and showed as CLLS graphs in section 2.2.

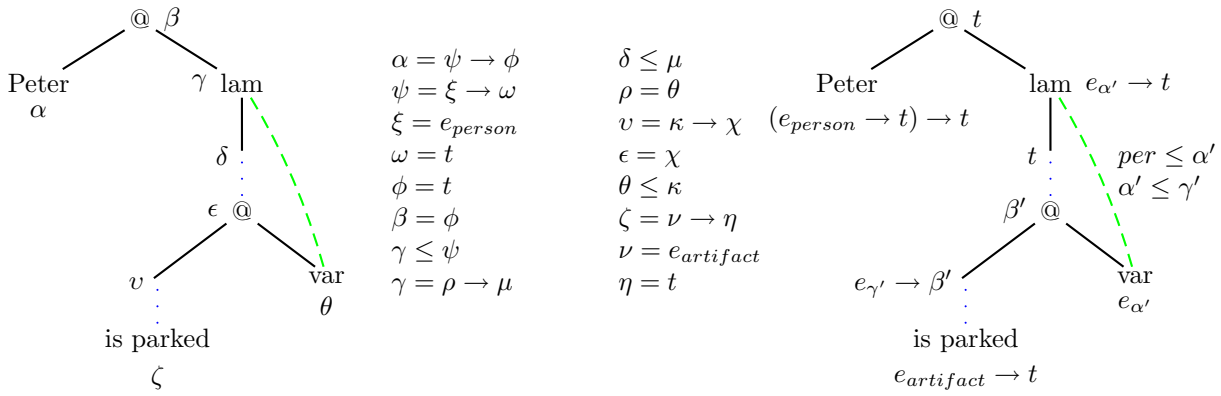


Figure 3: Example for an underspecified semantic structures with type constraints

The sentence "Peter is parked (out back)" yields the semantic structure (shown to the left in figure 3) and type constraints (shown in the middle), which can be combined in an annotated tree with only two additional constraints, which is shown to the right. Note that the variables that are annotated to the right hand graph are unrelated to the ones on the left, which we indicate by the prime. For a more detailed explanation of how constraint computing takes place, we refer to our technical report (Fuchss and Padó 2000).

3.4.3 Interpreting type constraints

Once that we have collected all type constraints, we can easily see if there is a type conflict for the literal reading - which would be the case if the corresponding constraints were inconsistent.

The literal reading is derived from the constraint by closing all dominance relations. Formally, checking for impending type conflicts is achieved by inserting new constraints into our type constraint set for the relaxed structure. These constraints try to close one dominance edge at a time by unifying the types of the upper and lower node. So the type check is an attempt to undo the relaxation. If all gaps and especially the relaxation gap can be closed without conflict, we assume that the sentence is just fine.

To apply this procedure to the example in the figure above: if we try to close the upper dominance edge, the additional constraint $\beta = t$ is added to the type constraints. This means no harm. But if we try to close the relaxation gap we opened above "is parked", we derive $(e_\gamma \rightarrow \beta) = (e_{artifact} \rightarrow t)$. We can infer $\gamma = artifact$ from this equation. This new constraint is inconsistent with what we already know: $(\gamma = artifact) \wedge (per \leq \alpha) \wedge (\alpha \leq \gamma)$ cannot be satisfied, as there is no sort that is at the same time a subsort of artifacts and a supersort of persons. So any solution of the type constraint must comprise additional material between υ and ζ, i.e. our sentence is in need of reinterpretation.

Now we can also motivate why we had to do relaxation in advance: If we had not done that, and there had been a type conflict, we would have derived two contradictory types for one node. By relaxing the structure, we "split" this node and avoided a clash.

3.4.4 Reinterpretation

Once we detect a type conflict, we can try to solve it by inserting an appropriate operator into the gap. Type theoretically speaking, the operator serves to make the types compatible.

At the moment, we use a database with a fixed number of reinterpretation operators. We test for all operators whether they fit into the gap by adding their type constraints to the existing constraint. The operators that fit are preselected. In the end, the user may choose one from the list of semantically feasible

operators, whose semantics is added to the semantics of the metonymical sentence. This yields a fully specified meaning for the expression.

In our example, the conflict is solved by a *owner-of* reinterpretation operator which introduces a function from a property P to the the property of being the owner of something with the property P. The semantics of this operator is of a from reminiscent of Dölling's reinterpretation templates (Dölling 1995):

$$\lambda P\, \lambda x_{per}\, .\, \exists y_{art}[owns(x, y) \wedge P(y)]$$

This operator has type $(e_{art} \rightarrow t) \rightarrow (e_{per} \rightarrow t)$, effectively solving the sortal conflict that occured in our example. It yields a sentence whose semantics could be phrased as "Peter owns a car that is parked (out back)."

4 Conclusion and further perspectives

We presented a method to complete the semantics of a reinterpretation case. We implemented this method in a component that can check underspecified semantical representations for type conflicts and solve those conflicts with the aid of reinterpretation operators stored in a database.

We concentrated on the type-conflicts-based detection of metonymical sentences. Our focus was on the process of reinterpretation, not on the generation of operators. The operators we use at the moment are from a hand-made database. To cover a broad range of metonymies, empirical research will be necessary (e.g. analysis of corpora). Nevertheless, the investigation of the process of reinterpretation can lead to new results concerning the properties of reinterpretation operators in a larger domain.

References

Dölling, J. (1995). Ontological domains, semantic sorts and systematic ambiguity. *International Journal of Human-Computer Studies 43*, 785–807.

Egg, M. (1999). Reinterpretation from a synchronic and diachronic point of view. Submitted. Available from http://www.coli.uni-sb.de/cl/projects/chorus/papers/egg99.html.

Egg, M. (2000). *Flexible semantic construction: the case of reinterpretation.* Habilitation thesis, Universität des Saarlandes.

Egg, M., J. Niehren, P. Ruhrberg, and F. Xu (1998). Constraints over lambda-structures in semantic underspecification. In *Proceedings of ACL/COLING '98*, Montreal, Canada, pp. 353–359.

Fuchss, R. and S. Padó (2000). Dokumentation Softwareprojekt Reinterpretation. Available from http://www.coli.uni-sb.de/~pado/ps/doku.ps.gz.

Koller, A., J. Niehren, and K. Striegnitz (2000). Relaxing underspecified semantic representations for reinterpretation. *Grammars 3*(2-3).

Schulte, C., G. Smolka, and J. Würtz (1994, May). Encapsulated search and constraint programming in Oz. In A. Borning (Ed.), *Second Workshop on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, vol. 874, Orcas Island, Washington, USA, pp. 134–150. Springer-Verlag.