Introduction to Computational Linguistics

UBG Parsing Exercises

---

1 Subsumption

As data structure to represent the feature structures, there are two choices:

1. the basic structures are the arcs
2. the basic structures are the nodes

For this purpose, we take the nodes (like in Carpenter 1992), because they make it easier to deal with reentrancies. Starting from this point, a feature structure can be defined recursively (similar to other directed graphs, e.g. trees):

> A feature structure is (represented by) a node from which labelled edges leave and lead to other feature structures. The node has a type value.

With this assumption, and the definition of feature structures from the lecture, we obtain the following recursive function.

Algorithm

A (typed) feature structure $F = \langle t, \epsilon \rangle$ subsumes a feature structure $G = \langle t', \epsilon' \rangle$, with $\epsilon, \epsilon' : \mathsf{Feat} \to \mathcal{F}$.

function subsumes(F, G)
if $\neg(t \sqsubseteq t')$ **then**
  return *false*
end if
for all $h \in \mathsf{Feat}$ where $\epsilon(h)$ is defined **do**
  if $\epsilon'(h)$ is not defined **then**
    return *false*
  else
    if subsumes($\epsilon(h)$, $\epsilon'(h)$) is *false* **then**
      return *false*
    end if
  end if
end for
return *true*

Does this deal with reentrancies correctly? No.

Adding reentrancies and checking for cycles

Let us start from an observation. We are not interested in the reentrancies of the second feature structure: that one can be more specific without affecting the subsumption relation. What makes a difference is that the first one has features or reentrancies more. We need to find out whether there are any features or any reentrancies in the first that are not in the second.

To do this, we extend the data structure for a node with a pointer field. This pointer is usually called *forward pointer*. If this pointer is used in a clever way, it is even enough to deal with *cyclic* feature structures, such as $\boxed{1}\left[\, f : \boxed{1} \,\right]$. A cyclic feature structure is a feature structure where there is at least one path that does not terminate, because it leads to itself. When we enter a node whose forward pointer has already been set, we only have to do the reentrancy check (why? think about this!! and maybe come up with an example!) and if this succeeds, we're done.

```
   function subsumes(F, G)
   if forward(F) is not defined then
3:    forward(F) := G
   else
      if forward(F) ≠ G then
6:       return false
      else
         return true
9:    end if
   end if
   if ¬(t ⊑ t′) then
12:   return false
   end if
   for all h ∈ Feat where ε(h) is defined do
15:   if ε′(h) is not defined then
         return false
      else
18:      if subsumes(ε(h), ε′(h)) is false then
            return false
         end if
21:   end if
   end for
   return true
```

For every node in the first feature structure, a pointer to the corresponding node in the second one is added. When a node in F should get such a pointer, but already has a *different* one, then the node was already visited, and we can return false. We exploit the equivalence classes to detect reentrancies.

Unification Algorithm (not on exercise sheet)

For unification, the algorithm is somewhat different, since we cannot completely follow the mathematical description, and of all feature structures that are subsumed by the unificands only pick the most general one. We need to *construct* the unification.

The basic idea of the algorithm: take the second feature structure, and add the information of the first one into it. The information means: the more specific types, the extra features and the extra reentrancies.

Note that the only place where this function can fail is in the type unification.

To treat the reentrancies (and the cycles) correctly, we need a more sophisticated forward pointer mechanism, called *dereference*. This term, together with the term *representative* stem from the *union-find* algorithm (see, e.g., `http://en.wikipedia.org/wiki/Disjoint-set_data_structure`), which is part of this unification algorithm.

Initially, every node in the feature structures of F and G is its own representative, which is achieved by letting the forward pointers point to the nodes itself. During unification, a node is dereferences by following the forward pointers until forward(Q) = Q.

```
    function unify(F, G)
    F := dereference(F); G := dereference(G)
 3: if F = G then
      return G
    end if
 6: forward(F) := G
    t' := unify(t, t');
    for all h ∈ feature-list(F) do
 9:   if h ∈ feature-list(G) then
        ε'(h) := unify(ε(h), ε'(h))
      else
12:     ε'(h) := ε(h)
      end if
    end for
15: return G
```

The edges to these revisited nodes need to be updated as well. This means that the nodes whose ϵ' leads to $\downarrow p(\mathrm{F})$, need to be modified such that it returns G instead of $\downarrow p(\mathrm{F})$. That can be solved either by storing ϵ'^{-1} information in the nodes (updating then consists of setting the $\epsilon'(h)$ of $\epsilon'^{-1}(h) \leftarrow \mathrm{G}$), or by adding another field $o : \mathsf{Feat} \rightarrow \mathcal{F}$ that indicates that the ϵ information is out-of-date, and that, in case $\epsilon(h)$ is needed, $o(h)$ is what shows the correct state.

For non-destructive unification, the feature structure needs to be copied first. That is a waste of time if the unification turns out to fail (like 95% of the unifications do).

2 Parsing

Modify a chart parsing algorithm to deal with unification-based grammars and discuss points of inefficiency. Try to present a solution (in pseudo-code or implemented form) for at least one of them.

As mentioned in the lecture, the algorithm itself does not need to be changed very much. The only change is that the category comparison in the chart needs to be done with subsumption checking, the rule application with unification. As an example, the original CYK algorithm is reproduced here.

```
    for all l = 2 to n do
      for all i = 0 to n - l do
3:      for all m = 1 to l - 1 do
          for all R_A → R_B R_C do
            if C(i, i+m, b) and C(i+m, i+l, c) then
6:            C[i, i+l, A] := true
            end if
          end for
9:      end for
      end for
    end for
```

Lines 5 and 6 are affected. Line 5 does a unification of the daughters in line 4 with chart edges. The (copied) result of that unification (A) is stored by the code in line 6. The indices are not affected. The third value in $\mathcal{C}()$ is a feature structure, which means that there will be a very large number of entries there. Before that is filled in, a subsumption test on the possible values in the third dimension of $\mathcal{C}$ needs to be done to be sure that the entry (or a more general form) is not there already. If it is, the new result can be discarded.

The following is the replacement for the lines 5 and 6:

```
    for all B in C(i, i+m, B) do
      for all C in C(i+m, i+l, C) do
3:      unify(R_B, B)
        unify(R_C, C) {Now A is instantiated}
        if A ≠⊥ then
6:        add new A to C(i, i+l, A)
        end if
      end for
9: end for
```

The function compatible is defined as follows:
compatible(F, G) := (unify(F, G) ≠ fail)