Towards "better" data structures: From atoms via compound terms to feature-structures

Detmar Meurers: Intro to Computational Linguistics I
OSU, LING 684.01, 26. February 2003

Overview

- From atoms to typed-feature structures
- What do non-atomic data structures represent?
- Combining non-atomic data structures
- Term unification
- Representing feature structures
- Feature structure unification

From atoms to typed-feature structures

- atoms, e.g. `verb_trans_first_sing_fin`
- compound terms, e.g.
  `verb(first,sing,fin,[noun(first,sing,nom,[]),noun(_,_,acc,[])])`
- feature structures, e.g.

```
category: verb,
vform: fin,
person: first,
number: sing,
subcat: [(category: noun, case:noun,
          person:first, number:sing, subcat: []),
         (category: noun, case:acc, subcat: [])]
```

- typed feature structures, e.g.

```
category: (verb,
           vform: (fin,
                   agr:(person: first,
                        number: sing)),
           subcat: [(category: (noun, case:nom,
                                agr: (person:first,
                                      number:sing),
                                subcat: [])),
                    (category: (noun, case:acc,
                                subcat: []))])
```

What do non-atomic data structures represent?

```
s --> np(Per,Num), vp(Per,Num).
```

The rule represents a set of ground instances, one for each possible **substitution** of a variable with a value.

```
s --> np(first,sing),  vp(first,sing).
s --> np(second,sing), vp(second,sing).
s --> np(third,sing),  vp(third,sing).
s --> np(first,plur),  vp(first,plur).
s --> np(second,plur), vp(second,plur).
s --> np(third,plur),  vp(third,plur).
```

Combining compound terms

```
np(third,sing) --> [he].
np(third,plur) --> [they].

vp(third,sing) --> [walks].
vp(third,plur) --> [walk].

s --> np(Per,Num), vp(Per,Num).
```

Two possible substitutions:

- `Per`=`third` and `Num`=`sing`
- `Per`=`third` and `Num`=`plur`

Most general unifiers

```
termUnify(f(X,h(Y,   e),g(d(Z),e)),
          f(Y,h(d(Z),e),g(Y,   e))).
```

Which substitution should one report?

- `X=d(a)`
- `X=d(b)`
- `X=d(h(a,b))`
- `...`

Compute the most general unifier (MGU):

- `X=d(Z)`

Infinite trees

- What happens when one unifies
  - `f(X)` with `X` ?
  - `f(a,g(X,b))` with `X` ?
- Add an **occurs check** to test whether the variable occurs in the term.
- In practice too costly.

Term unification

- A variable unifies with any term it does not occur in.
- An atom unifies only with an identical atom.
- Compound terms unify
  - if their functors are identical and
  - their arguments unify pairwise,

  and the substitutions obtained as a result of each of these unifications are compatible.

Converting predicates to lists (and vice versa) in Prolog

Prolog has a built-in predicate `=..` to relate predicates to lists, i.e.

```
Predicate =.. [Functor,Arg1,...,ArgN].
```

For example:

```
functor(a(b),c,d) =.. [functor,a(b),c,d].
```

This also works for zero-arity functors, i.e. constants:

```
a =.. [a].
```

Explicit term unification in Prolog

```
termUnify(X,Y) :-
   (var(X)
   ;var(Y)), !,
   X = Y.

termUnify(X,Y) :-
   X =.. [XFunctor|XArgs],
   Y =.. [YFunctor|YArgs],
   XFunctor=YFunctor,       % test for atom identity
   unifyArgs(XArgs,YArgs).

unifyArgs([],[]).
unifyArgs([X|Xs],[Y|Ys]) :-
   termUnify(X,Y),
   unifyArgs(Xs,Ys).
```

Representing feature structures in Prolog

- The operator ":" is defined to separate `feature:value`

```
?- op(500,xfy,:).
```

- Feature names and atomic values represented by Prolog atoms
- Prolog variables encode structure sharing.
- Feature structures represented as Prolog lists with open tails:
  - `[person:third, num:sing|_]`
  - `[person:X, num:sing, head:subj:person:X|_]`

Unifying feature structures in Prolog

```
unify(Dag,Dag) :- !.

unify([Feat:Val | Rest1], Dag) :-
   pathval(Dag,Feat,Val,Rest2),
   unify(Rest1,Rest2)

pathval([Feat:Val1 | Rest], Feat, Val2, Rest) :-
   !, unify(Val1,Val2).

pathval([Dag | Rest], Feat, Val, [Dag | Rest2]) :-
   pathval(Rest,Feat,Val,Rest2).
```

Towards grammar rules in PATR

```
rule(S,[NP,VP]) :-
  pathval(S,cat,s,_),
  pathval(NP,cat,np,_),
  pathval(VP,cat,vp,_),
  pathval(NP,per,X,_),
  pathval(VP,per,X,_),
  pathval(NP,num,Y,_),
  pathval(VP,num,Y,_).
```

Grammar rules in PATR

```
?- op(500,xfy,:).
?- op(500,xfx,--->).
?- op(600,xfy,===).

S ---> [NP,VP] :-
  S:cat  === s,
  NP:cat === np,
  VP:cat === vp,
  NP:per === X,
  VP:per === X,
  NP:num === Y,
  VP:num === Y.
```

Assigning a general meaning to "==="

```
X === Y :-
  denotes(X,Z),
  denotes(Y,Z).

denotes(Var,Var) :-
   var(Var),!.

denotes(Atom,Atom) :-
   atomic(Atom),!.

denotes(Dag:Feat,Value) :-
   pathval(Dag,Feat,Value,_).

pathval([Feat:Val1 | Rest], Feat, Val2, Rest) :-
   !, unify(Val1,Val2).
```

```
pathval([Dag | Rest], Feat, Val, [Dag | Rest2]) :-
   pathval(Rest,Feat,Val,Rest2).
```