Computational Linguistics II
— Grammars, Algorithms, Statistics —

Dan Flickinger

Oslo and Stanford Universities

danf@csli.stanford.edu

Tore Langholm

Universitetet i Oslo

torel@ifi.uio.no

Stephan Oepen

Oslo and Stanford Universities

oe@csli.stanford.edu

Review: Feature Structure Unification & Copying

Basic Notions

- Typed feature structures encoded as *directed acyclic graphs* (DAGs);
- each node bears a *type* and a set of *arcs* (aka feature – value pairs);
- feature structure reentrancy (coreference) corresponds to DAG identity;
- unification creates *equivalence classes*, encoded through *forwarding*.

Basic Operations

- *unify()* — make two DAGs equivalent, check and combine all information;

→ at each node, `glb()` types, forward, recurse over and accumulate arcs;

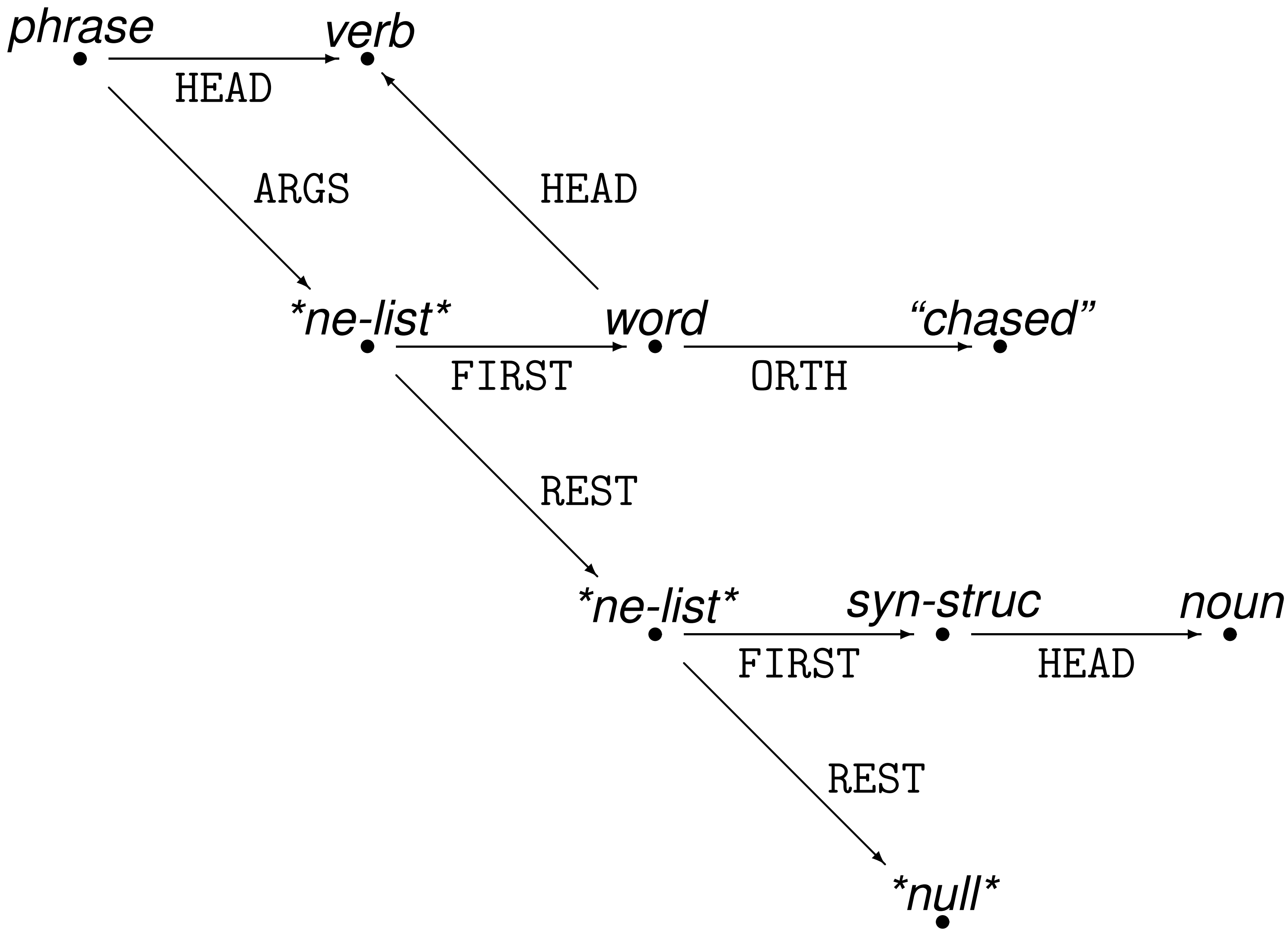
- *copy()* — create *structurally equivalent* copy (preserving reentrancies);

→ at each node, *copy* slot as short-term memory, reset upon completion.

Feature Structure Reentrancy (AVM)

$$\textit{phrase}\begin{bmatrix} \texttt{HEAD} & \boxed{1}\ \textit{verb} \\ \texttt{ARGS} & \textit{*ne-list*}\begin{bmatrix} \texttt{FIRST} & \textit{word}\begin{bmatrix}\texttt{ORTH} & \textit{“chased”}\\ \texttt{HEAD} & \boxed{1}\end{bmatrix} \\ \texttt{REST} & \textit{*ne-list*}\begin{bmatrix}\texttt{FIRST} & \textit{syn-struc}\begin{bmatrix}\texttt{HEAD} & \textit{noun}\end{bmatrix}\\ \texttt{REST} & \textit{*null*}\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

Feature Structure Reentrancy (DAG)



The Costs of Feature Structure Manipulation

Basic Cost Measure

- Visit each DAG node once (node operations ‘constant’): *full traversal*;
- *linear* in the number of nodes → upper bound is size of largest DAG.

Naïve Complexity Theory

- Prior to each (destructive) unification, make copies of both input DAGs;
- upon completion of each copy, recursively reset *copy* slot on all nodes.

`restore()`	`copy()`	`unify()`
1	2	5

The unify() vs. copy() Trade-Off

Destructive Unification [Boyer & Moore, 1972]

- Permanently alter both input dags: `setf()` on *forward*, *type*, and *arcs*;

→ *over copying* — two full copies required for only one result structure;

→ *early copying* — majority of unifications fail: many unnecessary copies.

Non-Destructive Unification [Wroblewski, 1987]

- Incrementally build up result DAG during unification, one node at a time;

→ eliminates over copying, reduces early copying more or less effectively.

Quasi-Destructive Unification [Tomabechi, 1991]

- Alter input DAGs in way that is reversible (at small cost): 'generations';

→ copy out result only after unification success, no over or early copying.

Generation Counting

- Protect DAG slots with *generation* counter → 'expiration date' of value;
- access: require valid generation; assignment: set value *and* generation;

→ implemented through interaction of global counter and ADT functionality.

```
(defstruct dag
  forward type arcs xcopy (generation 0))
(defparameter *generation* 1)
(defun dag-copy (dag)
  (when (= (dag-generation dag) *generation*) (dag-xcopy dag)))
(defsetf dag-copy dag-set-copy)
(defun dag-set-copy (dag value)
  (setf (dag-generation dag) *generation*)
  (setf (dag-xcopy dag) value))
```

Unification-Based Parsing

Adaptations to CFG-Based Chart Parser

- Make all elements of Σ, C, and P from the grammar feature structures;
- substitute *unification* and *equivalence test* for category comparison;
- unify category of passive edges with *argument position* of active edges;

$\rightarrow$ *edge* structure   LHS is DAG, RHS list of paths to argument positions;

$\rightarrow$ `fundamental-rule()`   result of unification is category for new edge;

$\rightarrow$ `pack-edge()`   equivalence test: two DAGs contain same information;

- test *spanning* passive edges for compatibility against start symbol S.

`#E[` *id* `:` `(`*i*–*j*`)` *dag* --> *edge*$_1$ ... *edge*$_i$. *path*$_{i+1}$ ... *path*$_n$ $\{$ *alternates* $\}^*$ `]`

```
#E[42: (0-8) head-specifier-rule --> 13 . (ARGS REST FIRST)]
```

Reminder: The Format of Grammar Rules in the LKB

$$\textit{mother}\begin{bmatrix}\texttt{HEAD} & \boxed{1} \\ \texttt{SPR} & \boxed{2} \\ \texttt{COMPS} & \langle\,\rangle \\ \ldots & \end{bmatrix} \longrightarrow \textit{daughter}_1\begin{bmatrix}\texttt{HEAD} & \boxed{1} \\ \texttt{SPR} & \boxed{2} \\ \texttt{COMPS} & \langle\boxed{3}\rangle\end{bmatrix}, \quad \textit{daughter}_2\;\boxed{3}\begin{bmatrix}\ldots\end{bmatrix}$$

$$\textit{mother}\begin{bmatrix}\texttt{HEAD} & \boxed{1} \\ \texttt{SPR} & \boxed{2} \\ \texttt{COMPS} & \langle\,\rangle \\ \ldots & \\ \texttt{ARGS} & \left\langle \textit{daughter}_1\begin{bmatrix}\texttt{HEAD} & \boxed{1} \\ \texttt{SPR} & \boxed{2} \\ \texttt{COMPS} & \langle\boxed{3}\rangle\end{bmatrix}, \quad \textit{daughter}_2\;\boxed{3}\begin{bmatrix}\ldots\end{bmatrix}\right\rangle\end{bmatrix}$$

Additional DAG Manipulation Functionality

Unification into Argument Position

- Additional parameter to `unify()`: unify *dag*$_2$ into *dag*$_1$ under *path*:

```
(defun unify (dag1 dag2 &optional path)
  ...)
```

- empty *path*: regular unification; otherwise find first *path* element in *dag*$_1$, recurse with corresponding *arc* value from *dag*$_1$, *dag*$_2$, and rest of *path*.

Equivalence Test

- Similar to `unify()`: traverse two dags in parallel, but no modifications;
- reentrancies: for each node, record corresponding node from second dag in *copy* slot; non-empty *copy* values need to match current nodes.

Unification-Based Parsing—Practical Concerns

Observations

- Typical systems: 90^{+} per cent of parsing time go to DAG manipulation;
- most unifications fail: predict unification failure cheaply, where possible;

→ *rule filter*: rule feeding relations; *quick check*: most likely failure paths;

- lexicalisation: argument positions in rules may be highly underspecified;

→ *head-driven* parsing: instantiate RHS bidirectionally, starting from head;

- many unifications fail very early: `copy()` more expensive than `unify()`;

→ memory is expensive: redo a couple of unfications instead of one copy.

Several orders of magnitude average speed-up by reducing constants

Unification-Based Parsing—Optimizations

Rule Filter

- ‘Specifier – Head’ cannot feed into first argument position of ‘Head – Complement’ (COMPS);

→ precompute *rule filter* relation;

- fundamental rule checks filter before attempting a unification.

Head-Driven Parsing

- First argument position of ‘Specifier – Head’ cannot fail: large number of active edges;

→ bi-directional rule instantiation: head argument position first.