A Comparative Introduction to XDG: The Linear Precedence Dimension in Action

Ralph Debusmann

and

Denys Duchier

Programming Systems Lab, Saarland University, Saarbrücken, Germany

and

Équipe Calligramme, LORIA, Nancy, France

This presentation

- adding the dimension of Linear Precedence (lp) to the example grammar
- new:
  - type definitions
  - one-dimensional principles (tree, valency, order)
  - multi-dimensional principles (climbing, barriers)
  - lexical classes

```
defdim lp {
...
}
```

Defining the new types

```
deftype "lp.label" {d
                    p
                    df n
                    mf vcf pf v vxf
                    root r}

deflabeltype "lp.label"
```

- also used as node labels on the LP dimension

```
deftype "lp.entry" {in: valency("lp.label")
                    out: valency("lp.label")
                    on: iset("lp.label")}

defentrytype "lp.entry"
```

Instantiating the lp principles

- re-used from the id dimension:
  - class of models: graph principle and tree principle
  - topological subcategorization: valency principle
- new:
  - constraining word order: order principle
  - use the solver for parsing: parse principle

Constraining the class of models

```
useprinciple "principle.graph" {
  dims {D: lp}}
```

```
useprinciple "principle.tree" {
  dims {D: lp}}
```

- parameter:
  - dimension: `D` (here: `lp`)

Constraining topological subcategorization

```
useprinciple "principle.valency" {
  dims {D: lp}
  args {In: _.D.entry.in
        Out: _.D.entry.out}}
```

- parameters:
  - dimension: `D` (here: `lp`)
  - in specification: `In` (here: lp lexical attribute `in`)
  - out specification: `Out` (here: lp lexical attribute `out`)

Constraining word order

```
useprinciple "principle.order" {
  dims {D: lp}
  args {On: _.D.entry.on
        Order: [d
                p
                df n
                mf vcf pf v vxf
                root r]
        Projective: true}}
```

- parameters:
  - dimension: `D` (here: `lp`)
  - on specification: `On` (here: lp lexical attribute `on`)
  - total order on the set of edge labels: `Order`
  - projectivity constraint: `Projective`

Use the solver for parsing

```
useprinciple "principle.parse" {
  dims {D: lp}}
```

- parameter:
  - dimension: `D` (here: `lp`)
- if not used, the solver regards the input as a bag of words
- useful for debugging (e.g. generate all licensed linearizations)
- demo!

Introducing the multi dimension

- convenience dimension for multi-dimensional principles
- hold certain lexical features and/or node attributes
  - `blocks_lpid`
- instantiate multi-dimensional principles:
  - restrict the class of models: climbing principle
  - impose restrictions on climbing: barriers principle
- models: graphs without edges

Restricting the class of models

```
useprinciple "principle.climbing" {
  dims {D1: lp
        D2: id}}
```

- parameters:
  - dimensions: `D1`, `D2` (here: `lp`, `id`)
  - the lp dimension is a flattening of the id dimension

Imposing restrictions on climbing

```
useprinciple "principle.barriers" {
  dims {D1: lp
        D2: id
        Multi: multi}
  args {Blocks: _.Multi.entry.blocks_lpid}}
```

- parameters:
  - dimensions: `D1`, `D2`, `Multi` (here: `lp`, `id`, `multi`)
  - arguments: `Blocks` (here: lexical attribute `blocks_lpid` on the multi dimension)

Lexicon

- lexical classes
  - new lexical classes to specify lp and id/lp properties
  - update existing lexical classes to inherit from them
- lexical entries
  - apply the updated lexical classes

Defining new lexical classes: cnoun_lp

```
defclass "cnoun_lp" {
  dim lp {in: {mf? root?}
          out: {df?}
          on: {n}}
  dim multi {blocks_lpid: {det}}}
```

- *a common noun can land in the Mittelfeld or can be root, offers a determiner field, has node label* $\mathtt{n}$*, and blocks its determiner from climbing up*

$$\mathsf{df} \prec \mathsf{n} \prec \mathsf{mf} \prec \mathsf{root}$$

Defining new lexical classes: fin_lp

- *verb 2nd position:* ***rich topological domain***

```
defclass "fin_lp" {
  dim lp {in: {root?}
          out: {mf* vcf? vxf?}
          on: {v}}
  dim multi {blocks_lpid: {subj obj vbse vprt vinf part}}}
```

$$\mathsf{mf} \prec \mathsf{vcf} \prec \mathsf{v} \prec \mathsf{vxf} \prec \mathsf{root}$$

Defining new lexical classes: can and noncan

- *canonical position:* ***impoverished topological domain***

```
defclass "can" {
  dim lp {in: {vcf? root?}
          on: {v}
          out: {vcf?}}}
```

- *non-canonical position:* ***rich topological domain***

```
defclass "noncan" {
  dim lp {in: {vxf? root?}
          on: {v}
          out: {mf* vcf? vxf?}}}
```

$$\mathrm{mf} \prec \mathrm{vcf} \prec \mathrm{v} \prec \mathrm{vxf} \prec \mathrm{root}$$

Updating lexical classes: cnoun

```
defclass "cnoun" Word Agrs {
  "cnoun_id"
  "cnoun_lp"
  dim id {agrs: Agrs}
  dim lex {word: Word}}
```

- *a common noun inherits from the classes for common nouns on the id and lp dimensions, has agreements `Agrs` and word form `Word`*

Updating lexical classes: fin

```
defclass "fin" Word {
  "fin_id"
  "fin_lp"
  dim lex {word: Word}}
```

- *a finite verb noun inherits from the classes for finite verbs on the id and lp dimensions, and has word form* `Word`

Updating lexical classes: mainverb

```
defclass "mainverb" Word1 Word2 Word3 {
    "fin" {Word: Word1}
  | ("vbse" {Word: Word2} & "can")
  | ("vprt" {Word: Word3} & "can")
  | ("vinf" {Word: Word2} & ("can" | "noncan"))}
```

- *a mainverb is either finite (word form `Word1`), a bare infinitive (`Word2`) in canonical position, a past participle (`Word3`) in canonical position, or a zu-infinitive (`Word2`) in either canonical or non-canonical position*

Applying the updated lexical classes

```
defentry {
  "cnoun" {Agrs: {nom acc}
           Word: "frau"}}

defentry {
  "transitive"
  "mainverb" {Word1: "liebt"
              Word2: "lieben"
              Word3: "geliebt"}}
```

- lexical entries need not be changed