A Comparative Introduction to XDG: The Immediate Dominance Dimension in Action

Ralph Debusmann

and

Denys Duchier

Programming Systems Lab, Saarland University, Saarbrücken, Germany

and

Équipe Calligramme, LORIA, Nancy, France

This presentation

- introduce the XDG Development Kit (XDK)
- the XDK provides:
  - facilities to cook your own multi-dimensional grammar formalism
  - metagrammar facilities to organize the lexicon effectively
- implement an example grammar fragment made up of the dimensions id and lex

XDK: Making life easy

- for grammar writers
- adopts a software engineering perspective to ease grammar engineering:
  - modules (dimensions, principles, lexical abstractions)
  - re-usability
  - composition

Structuring the lexicon

- unfeasible to write the individual lexical entries directly
- abstractions: lexical classes
- combining the descriptions:
  - conjunction (inheritance)
  - disjunction (alternations)
- descriptions compiled into individual lexical entries
- two goals:
  - improve grammar engineering
  - enable the statement of linguistic generalizations

Defining a dimension

```
defdim id {
  %% %%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %% define types
    ...
  %% %%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %% use principles
    ...
}
```

Defining types

- XDK: provides static typing to ensure the coherence of the grammar, ease debugging:
  - need to define types
- for each dimension
- e.g. domains, type constructors for sets, records etc.
- three special types picked out for each dimension:
  - edge labels
  - lexical entry record
  - node attributes record
- each node:

$$\begin{bmatrix} \mathsf{entry} & : & \ldots \\ \mathsf{attrs} & : & \ldots \end{bmatrix}$$

Defining types contd.

- XDK: multi-dimensional
- i.e. for each node:

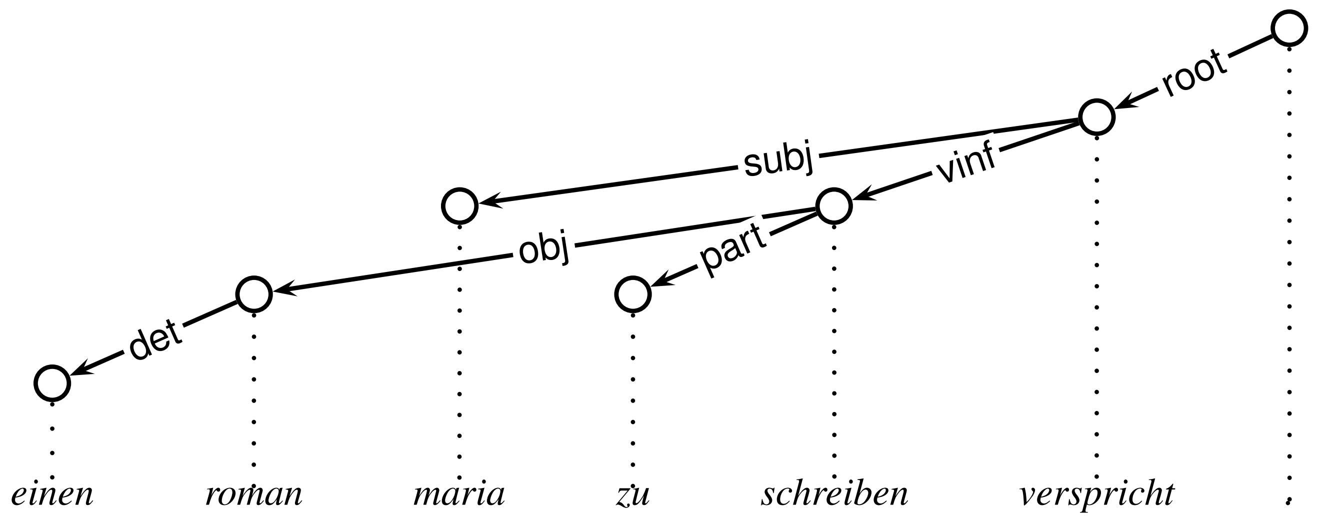
$$\begin{bmatrix} \mathsf{dim}_1 & : & \begin{bmatrix} \mathsf{entry} & : & \dots \\ \mathsf{attrs} & : & \dots \end{bmatrix} \\ \dots & & \\ \mathsf{dim}_n & : & \begin{bmatrix} \mathsf{entry} & : & \dots \\ \mathsf{attrs} & : & \dots \end{bmatrix} \end{bmatrix}$$

Defining domain types

```
deftype "id.label" {det subj obj vbse vprt vinf part root}
```

```
deftype "id.agr" {nom acc}
```

- root: additional root node for convenience:



Defining valency types

```
valency("id.label")
```

- specialized type constructor for the valency principle (subcategorization):

$$\mathit{Roman} \quad : \quad \begin{bmatrix} \mathsf{in} & : & \{\mathsf{subj}?, \mathsf{obj}?, \mathsf{iobj}?\} \\ \mathsf{out} & : & \{\mathsf{det}!, \mathsf{adj}*, \mathsf{prep}?, \mathsf{rel}?\} \end{bmatrix}$$

Defining set types

- idea: having two sets, how to combine them? (inheritance)
- two natural possibilities:
  - intersection
  - union
- we want both possibilities for different kinds of sets:
  - intersective sets
  - accumulative sets

Set types: Examples

- set of possible agreements:

```
deftype "id.agrs" iset("id.agr")
```

$$\{\mathsf{nom}, \mathsf{acc}\} \cap \{\mathsf{acc}\} = \{\mathsf{acc}\}$$

- set of agreeing edge labels:

```
deftype "id.agree" set("id.label")
```

$$\{\mathsf{det}\} \cup \{\mathsf{adj}\} = \{\mathsf{det}, \mathsf{adj}\}$$

Defining record types

```
deftype "id.attrs" {agr: "id.agr"}

deftype "id.entry" {in: valency("id.label")
                    out: valency("id.label")
                    agrs: "id.agrs"
                    agree: set("id.label")
                    govern: map("id.label" "id.agrs")}
```

Defining map types

```
map("id.label" "id.agrs")
```

- functional type
- shorthand for records having the same type at each feature

$$\begin{bmatrix} f_1 & : & t \\ \dots & & \\ f_n & : & t \end{bmatrix}$$

Picking out the special types

- edge labels domain:

```
deflabeltype "id.label"
```

- lexical entry subrecord:

```
defentrytype "id.entry"
```

- node attributes subrecord:

```
defattrstype "id.attrs"
```

Instantiating the principles

- well-formedness conditions
- principles can be:
  - one-dimensional
  - multi-dimensional
- taken from an extensible principle library
- parametrized

Constraining the class of models

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

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

- parameters:
  - dimension: `D` (here: `id`)

Constraining subcategorization

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

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

Constraining case assignment

- idea: assign a case to each node
- lexically? too uneconomical
- optimization: lexically assign a set of possible cases and pick out one of these for each node:

$$\forall v \in V \quad : \quad \mathsf{agr}(v) \in \mathsf{agrs}(v)$$

- generalized, parametric agr principle from the XDK principle library:

$$\forall v \in V \quad : \quad \mathsf{F}_1(v) \in \mathsf{F}_2(v)$$

```
useprinciple "principle.agr" {
  dims {D: id}
  args {Agr: _.D.attrs.agr
        Agrs: _.D.entry.agrs}}
```

Constraining case agreement

- idea: heads can make certain dependents agree with them
- e.g. in German the determiner and adjective dependents of nouns must agree with the nouns:

$$\forall h \xrightarrow{l} d \quad : \quad l \in \mathsf{agree}(h) \;\Rightarrow\; \mathsf{agr}(h) = \mathsf{agr}(d)$$

- generalized, parametric agreement principle from the XDK principle library:

$$\forall h \xrightarrow{l} d \quad : \quad l \in \mathsf{F}_1(h) \;\Rightarrow\; \mathsf{F}_2(h) = \mathsf{F}_3(d)$$

```
useprinciple "principle.agreement" {
  dims {D: id}
  args {Agr1: ^.D.attrs.agr
        Agr2: _.D.attrs.agr
        Agree: ^.D.entry.agree}}
```

Constraining case government

- idea: heads can govern the case of certain dependents
- e.g. in German and English, finite verbs require their subjects to be nominative:

$$\forall h \xrightarrow{l} d \quad : \quad \mathsf{agr}(d) \in \mathsf{govern}(h)(l)$$

- generalized, parametric government principle from the XDK principle library:

$$\forall h \xrightarrow{l} d \quad : \quad \mathsf{F}_1(d) \in \mathsf{F}_2(h)(l)$$

```
useprinciple "principle.government" {
  dims {D: id}
  args {Agr2: _.D.attrs.agr
        Govern: ^.D.entry.govern}}
```

Defining the lex dimension

- special dimension
- models: graphs without edges
- purpose: assign a word form to each lexical entry

```
defdim lex {
  defentrytype {word: string}
}
```

Example lexical classes

```
defclass "fin_id" {
  dim id {in: {root?}
          out: {subj!}
          govern: {subj: {nom}}}}
```

- *a finite verb can optionally be root, and requires a subject in nominative case*

```
defclass "transitive" {
  dim id {out: {obj!}
          govern: {obj: {acc}}}}
```

- *a transitive verb requires an object in accusative case*

Using inheritance, parameters and conjunction

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

- definition of a parametric class: parameter `Word`
- *a finite verb inherits from the class of finite verbs for the id dimension, and has word form* `Word`

Using parameters and disjunction

```
defclass "mainverb" Word1 Word2 Word3 {
    "fin" {Word: Word1}
  | "vbse" {Word: Word2}
  | "vprt" {Word: Word3}
  | "vinf" {Word: Word2}}
```

- instantiation of a parametric class
- *a main verb is either finite (word form `Word1`), a bare infinitive (`Word2`), a past participle (`Word3`), or a zu-infinitive (`Word2`)*

Defining lexical entries

- idea: describe how to actually generate lexical entries

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


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

- `defentry`-expressions describe a set of lexical entries
- use the same language as for lexical classes
- must be assigned a word form