A Principle Compiler for Extensible Dependency Grammar

Bachelor Thesis
Programming Systems Lab

Jochen Setz, 08.11.2007

Betreuer: Ralph Debusmann

Introduction

- XDG is a contraint-based Meta-Grammarformalism
- Models are formalised by Constraint in First-Order Logic $\Rightarrow$ Principles

Introduction

- XDG is a contraint-based Meta-Grammarformalism
- Models are formalised by Constraint in First-Order Logic $\Rightarrow$ Principles
- XDK is the implementation of XDG
- Principles in XDK are Mozart/Oz-Constaints on finit sets

Introduction

- XDG is a contraint-based Meta-Grammarformalism
- Models are formalised by Constraint in First-Order Logic $\Rightarrow$ Principles
- XDK is the implementation of XDG
- Principles in XDK are Mozart/Oz-Constaints on finit sets

$\Rightarrow$ Developing new Principles is not trivial in XDK

Bachelor's Thesis

- Developing a Programm (PrincipleWriter) which translates FOL-Constranints in Mozart/Oz-Constraints.
- Closing the gap between the formalisation and the implementation.
- Enable more users to write new Principles.
- Speed up writing of grammars.
- Raise up the attractiveness of the development System.

Contents

- XDG
- XDK
- PrincipleWriter
  - PrincipleWriter User Language
  - Type checking/Type inference
  - Semantics
  - Optimizing
- Conclusions

XDG

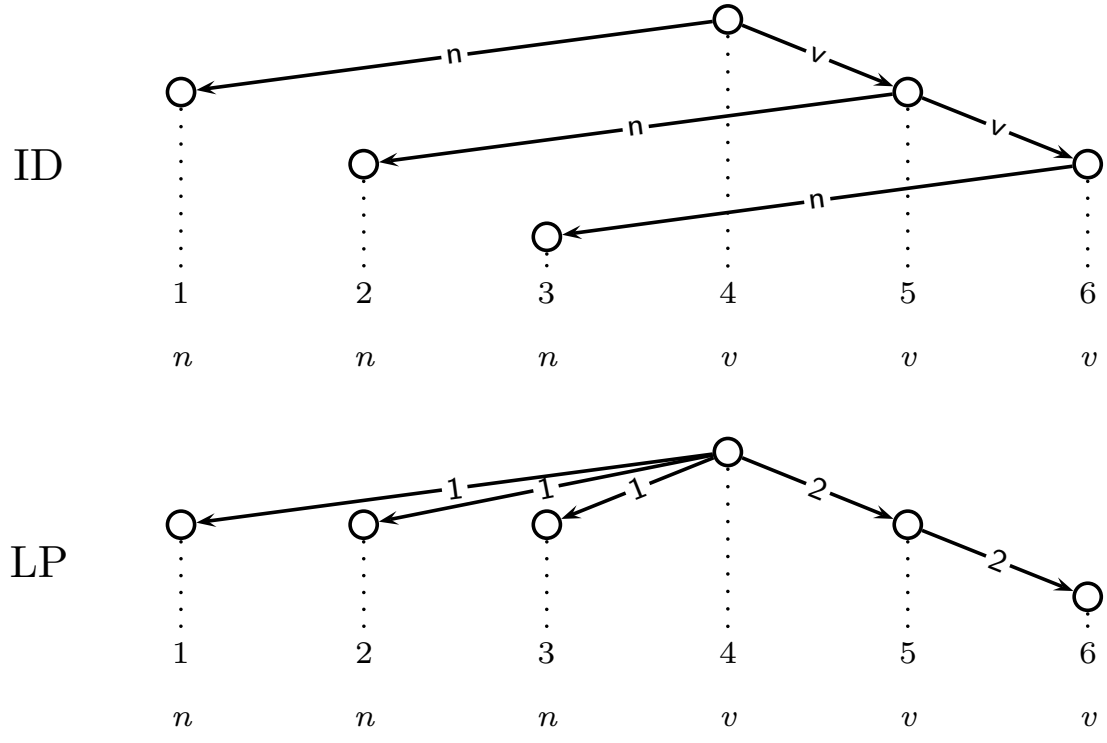
- XDG is a grammarformalism based on dependency grammar.
- XDG grammars are extensible by arbitrary linguistic aspects like word order
- Each aspects works on own Dimensions
- Developing of grammars becomes more easy.
- The models of a grammar are represented by multigraphs.

XDK: Multigraphs

Exists of

- a finite set of nodes (1,.....,n)
- a finit set of labeled edges
- a strict order $< \subseteq V \times V$
- a node-word-mapping $nw \in V \longrightarrow V$
- a node-attribute-maping $na \in V \longrightarrow D \longrightarrow A \longrightarrow U$

XDG: Mulitraphs (con'd)



Each node contains additionally attributes.

XDG: Grammars

A grammar in XDG exists of

- a Multigrphtype MT (all possible Dimensions, Words, Edgelabels and attributes)
- a Lexicon lex
- a set of Principles

XDG: Example CSD

- Cross-Serial Dependencies

$$\text{CSD} = \{n^1 \ldots n^k v^1 \ldots v^k \mid k \geq 1\}$$

- Each n and v is associated with an index i.
- It must hold, that each n_i must be an argument of v_i.

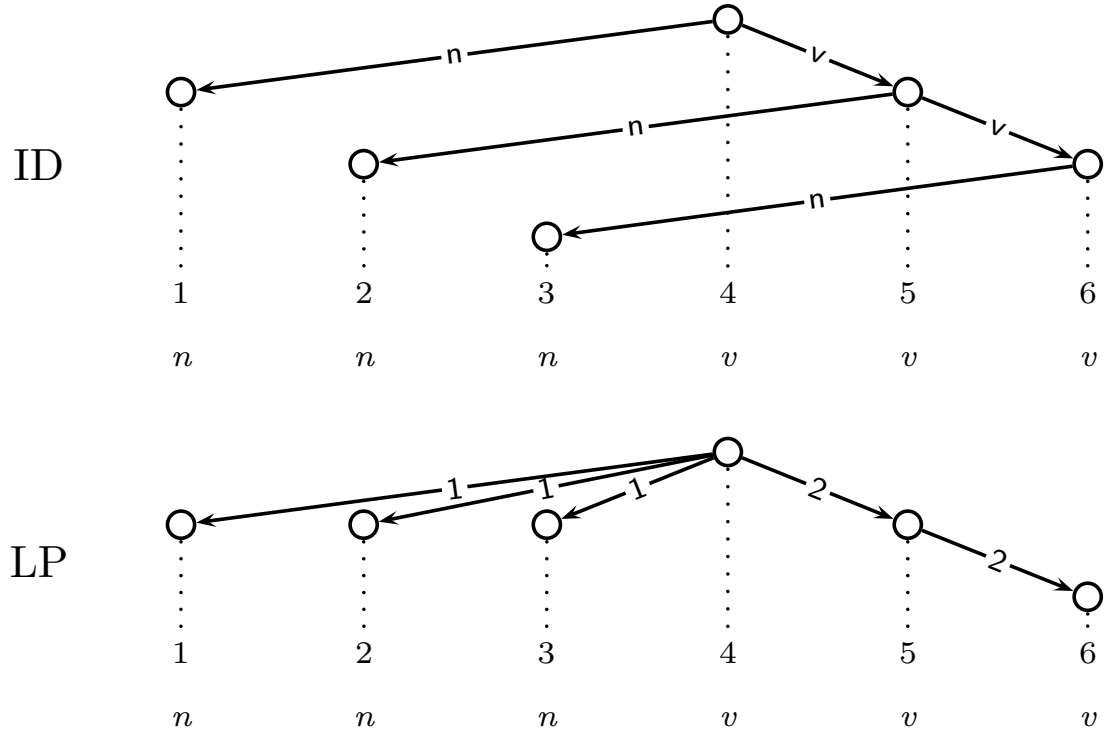
(omdat)	*ik*	*Cecilia*	*Henk*	*de nijlpaarden*	*zag*	*helpen*	*voeren*
(that)	*I*	*Cecilia*	*Henk*	*the hippos*	*saw*	*help*	*feed*

"(that) I saw Cecilia help Henk feed the hippos"

XDG: Example CSD (Con'd)

- Mulitgraphtype MTCSD = (D,W,L,dl,A,T,dat).
- Dimensions ID (Immediate Dominance) and LP (Linear Precedence).
- Words n and v
- Labels: n,v (ID) and 1,2 (LP)
- Attributes: in, out, order

XDG: Example CSD (con'd)

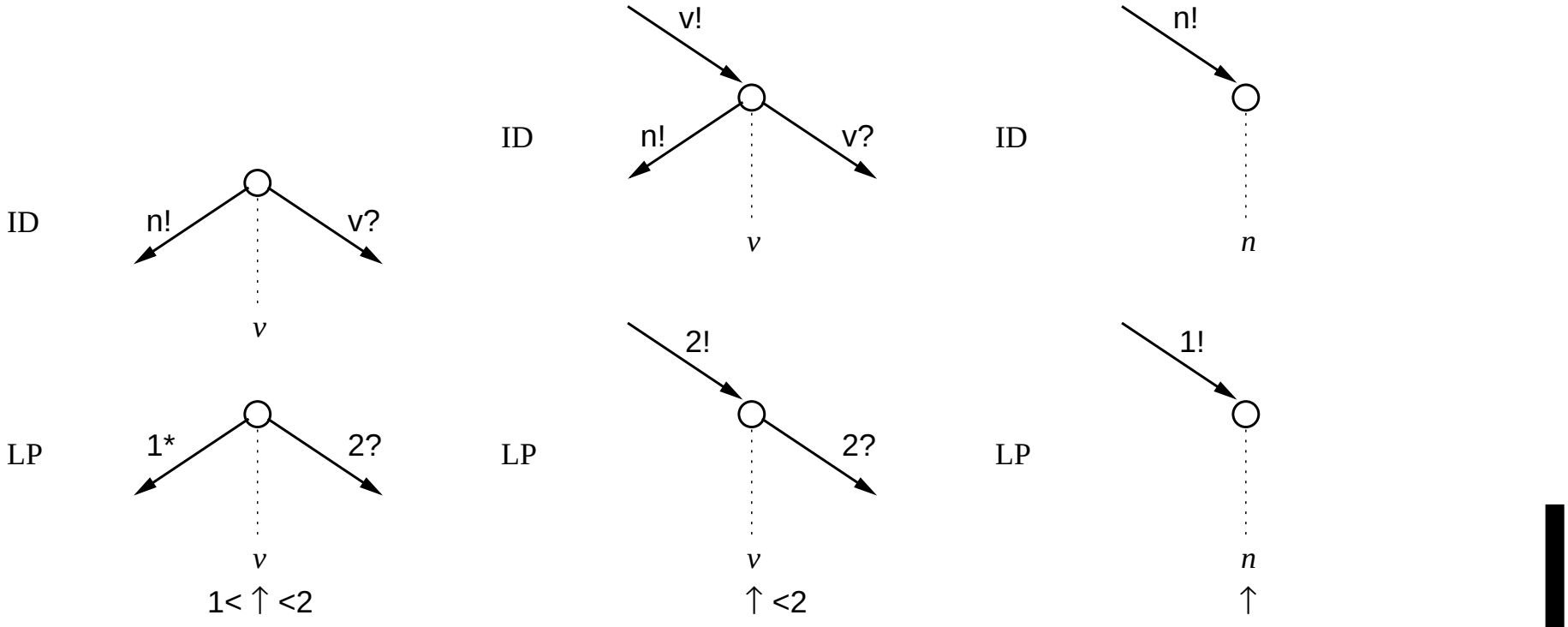


XDG: Lexicon

- Maps word to sets of lexial entries.
- A lexical entry specifies the values of the lexial attributes for each dimension.

XDG: Lexicon

- Maps word to sets of lexial entries.
- A lexical entry specifies the values of the lexial attributes for each dimension.



XDG: Principles

- State the well-formedness conditions of the multigraph.
- A grammar writer can create new principles or select existing ones from the principle library.
- Principles are FOL-Constraints, which abstracts over dimensions.

XDG: Principles

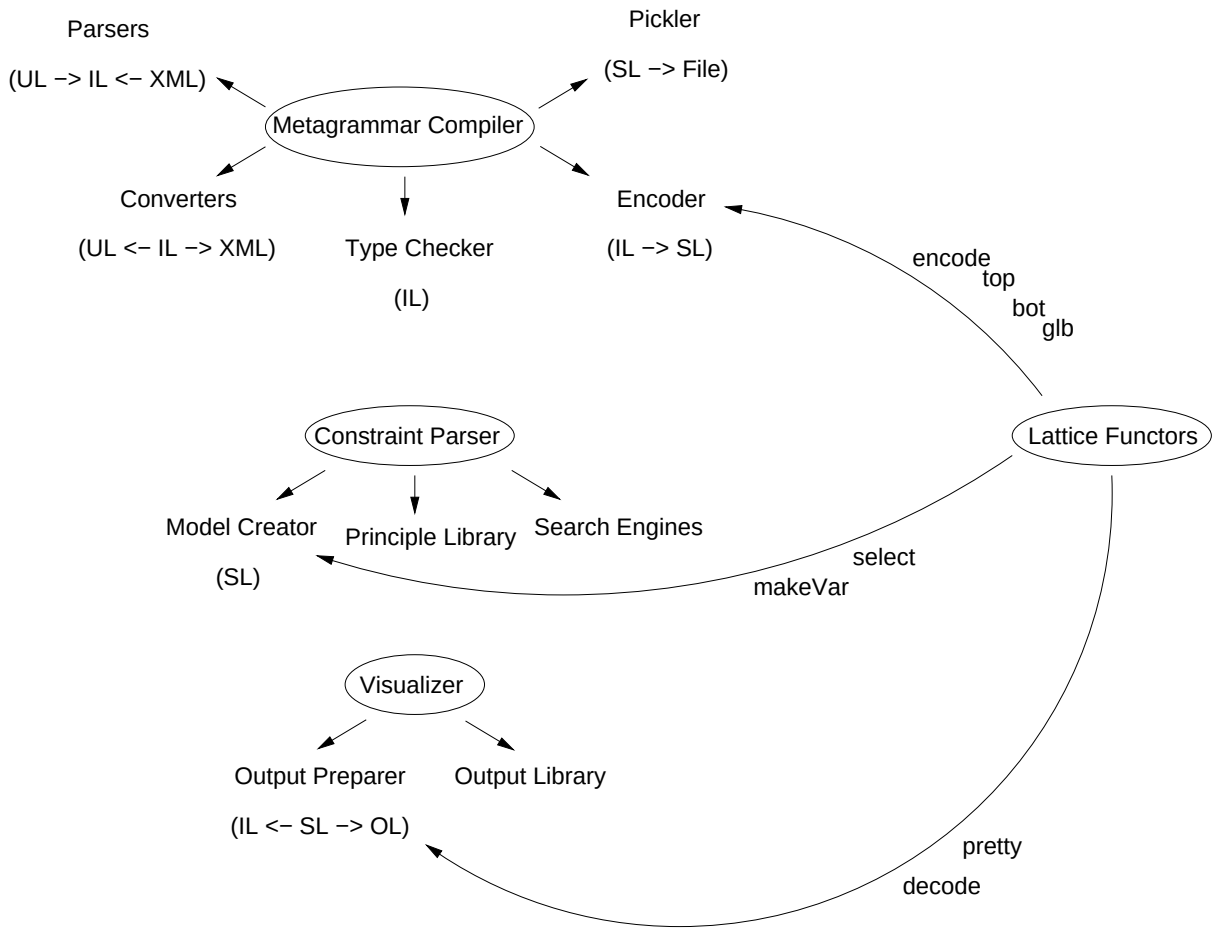
- State the well-formedness conditions of the multigraph.
- A grammar writer can create new principles or select existing ones from the principle library.
- Principles are FOL-Constraints, which abstracts over dimensions.
- i.E. Tree-Principle (One root, zero or one mother, no cycles, disjunct subtrees.)

XDG: Principles CSD

- The CSD grammar uses:
  - ID: Tree, Valency, CSD
  - LP: Tree, Valency, Order
  - ID and LP: Climbing
- The CSD-Principle states, that all n-dependents of a verb v must follow after the n-dependents of the verbs over v.

$$csd_d = \forall v, v' :$$
$$v \xrightarrow{n}_d v' \Rightarrow \forall v'', v''' : v'' \rightarrow_d^+ v \wedge v'' \xrightarrow{n}_d v''' \Rightarrow v''' < v'$$

XDK: Architektur



XDK: Description Language

- Describes the multigraphtype.
- Describes the lexicon.
- The XDK DL cannot describe principles! DL can just select existing ones.
- New Principles have to be inserted into the principle library.

XDK: Description Language

- Describes the multigraphtype.
- Describes the lexicon.
- The XDK DL cannot describe principles! DL can just select existing ones.
- New Principles have to be inserted into the principle library.

Grammatik G = (MT, lex, P)

↑ ↑ ↑

✔ ✔ ✗

XDK: Multigraphtyp (CSD)

- Mulitgraphtype definition of the MTCSD for dimension id:

```
defdim id {
  deftype "id.label" {n v}
  deflabeltype "id.label"
  defentrytype {in: tuple("id.label" {"!","?","+","*"})
                out: tuple("id.label" {"!","?","+","*"}}

  [ ... ]
}
```

XDK: Lexicon (CSD)

- DL offers lexical classes which represents lex. entries which abtract over variables.
- And it offers pure lexical entries.
- By converting from UL to IL, the lex. classes are transformed to lexical entries.

XDK: Lexicon (CSD)

- DL offers lexical classes which represents lex. entries which abtract over variables.
- And it offers pure lexical entries.
- By converting from UL to IL, the lex. classes are transformed to lexical entries.
- Example: The verbes in CSD:

```
defclass "verb" Word {
  dim id {out: {n! v?}}
  dim lp {out: {"1"* "2"?}
          order: <"1" "^" "2">}
  dim lex {word: Word}}
```

```
defentry {
  "verb" {Word: "v"}
  dim lp {out: {"1"*}}}
```

```
defentry {
  "verb" {Word: "v"}
  dim id {in: {v!}}
  dim lp {in: {"2"!}}}
```

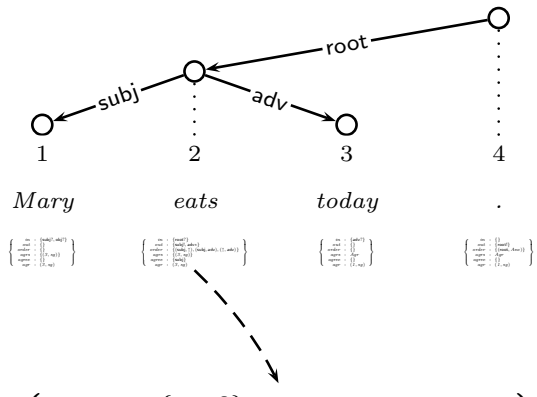
XDK: Lexicon (CSD) (Con'd)

```
defentry {
  dim id {in: {}
          out: {n! v?}}
  dim lp {in: {}
          out: {"1"* "2"?}
          order: <"1" "^" "2">}
  dim lex {word: "v"}}

defentry {
  dim id {in: {v!}
          out: {n! v?}}
  dim lp {in: {"2"!}
          out: {"2"?}
          order: <"^" "2"}
  dim lex {word: "v"}}

defentry {
  dim id {in: {n!}}
  dim lp {in: {"1"!}}
  dim lex {word: "n"}}
```

XDK: Principles

- Multigraphs are modelled in IL with finite sets of $\mathbb{N}$
- The IL representation is a Mozart/Oz record with Sets as values.
- Principles in XDK must be Mozart/Oz Constraints over the record repectively the sets.

XDX: Mulitgraph



$$\left\{\begin{array}{rcl} in & : & \{\text{root?}\} \\ out & : & \{\text{subj!}, \text{adv}*\} \\ order & : & \{(\text{subj}, \uparrow), (\text{subj}, \text{adv}), (\uparrow, \text{adv})\} \\ agrs & : & \{(3, sg)\} \\ agree & : & \{\text{subj}\} \\ agr & : & (3, sg) \end{array}\right\}$$

```
o(index: 2
  word: eats
  nodeSet: {1 2 3 4}#4
  model: o(mothers: {4}#1
           daughters: {1 3}#2
           up: {4}#1
           down: {1 3}#2
           index: 2
           eq: {2}#1
           equp: {2 4}#2
           eqdown: {1 2 3}#3
           labels: {5}#1
           mothersL: o(adv: {}#0
                       root: {4}#1
                       subj: {}#0)
           daughtersL: o(adv: {3}#1
                         root: {}#0
                         subj: {1}#1)
           upL: o(adv: {}#0
                  root: {4}#1
                  subj: {}#0)
           downL: o(adv: {3}#1
                    root: {}#0
                    subj: {1}#1)))
```

XDK: Principles (Con'd)

- A Principle exists of two parts: the prinziple definition and a set of node-constraint-functors.
- The prinziple definition defines dimensions-variables over those the principle abtracts, and the used node-constraint-functors.
- Example: csd-principle:

```
defprinciple "principle.csd" {
  dims {D}
  constraints {"CSD": 110}}
```

XDK: Principles (Con'd)

- The node-constraint-funktors are Mozart/Oz procedures, with three arguments:
  - List of noderecords
  - The grammar
  - A funktion which maps dimension-variables to dimensions

XDK: Principles (Con'd)

- Example: csd-principle:

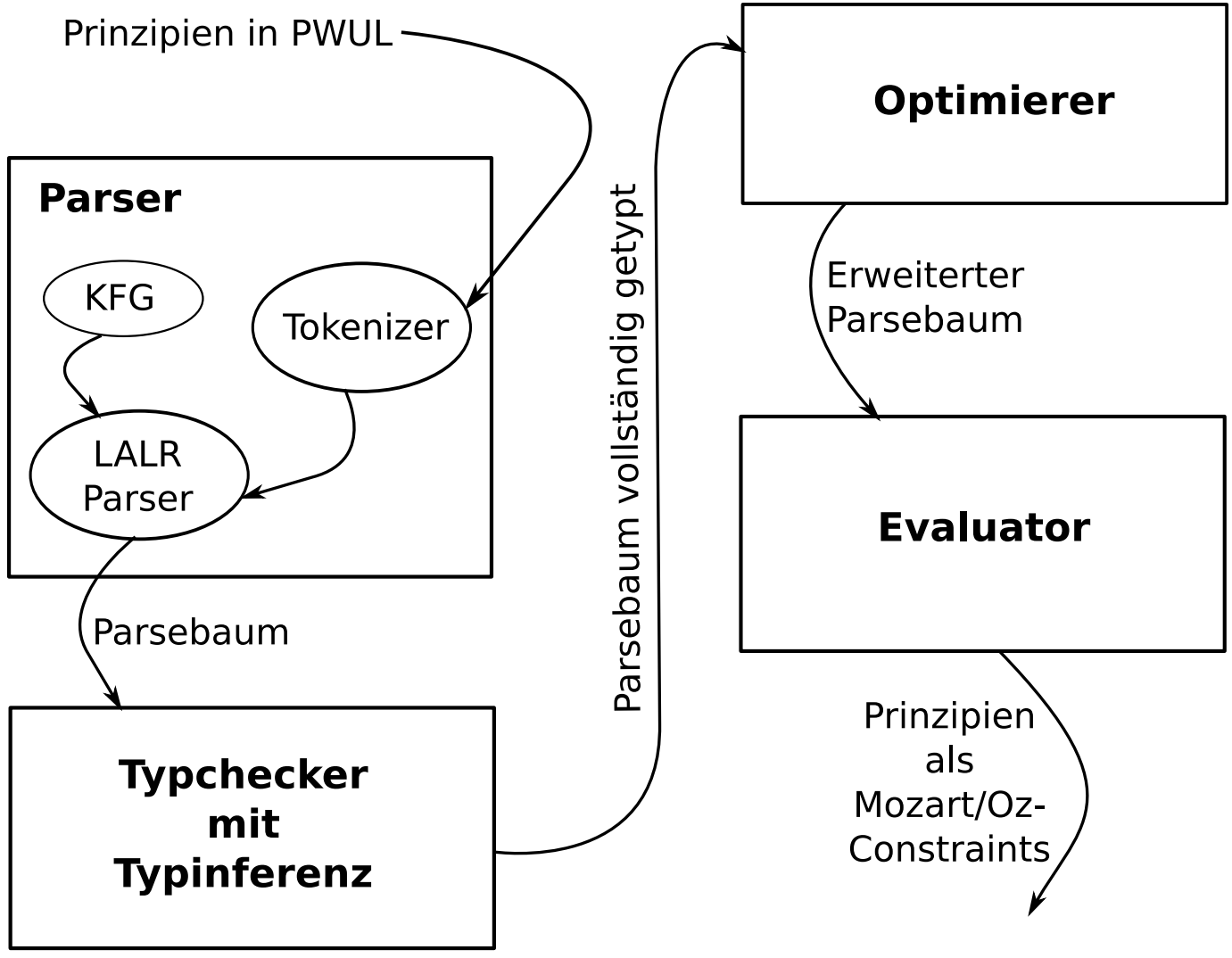
```
(  1) proc {Constraint Nodes G GetDim}
(  2)    DIDA = {GetDim 'D'}
(  3)    PosMs = {Map Nodes
(  4)             fun {$ Node} Node.pos end}
(  5) in
(  6)    for Node in Nodes do
(  7)       NDaughtersMs =
(  8)       {Map Nodes
(  9)        fun {$ Node} Node.DIDA.model.daughtersL.n end}
( 10)       NDaughtersUpM = {Select.union NDaughtersMs Node.DIDA.model.up}
( 11)       PosNDaughtersUpM = {Select.union PosMs NDaughtersUpM}
( 12)       NDaughtersM = Node.DIDA.model.daughtersL.n
( 13)       PosNDaughtersM = {Select.union PosMs NDaughtersM}
( 14)    in
( 15)       {FS.int.seq [PosNDaughtersUpM PosNDaughtersM]}
( 16)    end
( 17) end
```

- Far away from the formalisation in FOL!

PrincipleWriter

- Formalisation (XDG) $\Leftrightarrow$ Implementation (XDK)
- Multigraphtype and lexicon follows the formalisation, the principles are much harder to implement:
  - FOL must be translated in Mozart/Oz constraints.
  - Experts in Mozart/Oz are needed.
  - specially for the optimisation.
- The PrincipleWriter closes this gap. Principles can now be written as FOL constraints!

PW: Architecture



PW: User Language

XDG	PWUL	XDG	PWUL	XDG	PWUL
$\neg$	~	$=$	=	$V1 \rightarrow_D V2$	edge(V1 V2 D)
$\wedge$	&	$\neq$	~	$V1 \overset{L}{\rightarrow}_D V2$	edge(V1 V2 L D)
$\vee$	|	$\in$	in	$V1 \rightarrow^*_D V2$	domeq(V1 V2 D)
$\Rightarrow$	=>	$\notin$	notin	$V1 \rightarrow^+_D V2$	dom(V1 V2 D)
$\Leftrightarrow$	<=>	$\subseteq$	subseteq	$V1 \overset{L}{\rightarrow}^+_D V2$	dom(V1 V2 L D)
$\forall$	forall	$\parallel$	disjoint	$<$	<
$\exists$	exists	$\cap$	intersect	$w(v) = w$	v.word = w
$\exists!$	existsone	$\cup$	union		
		$\setminus$	minus		

PW: Example CSD

The CSD-principle in FOL:

$$csd_d = \forall v, v' :$$
$$v \xrightarrow{n}_d v' \Rightarrow \forall v'', v''' : v'' \rightarrow_d^+ v \wedge v'' \xrightarrow{n}_d v''' \Rightarrow v''' < v'$$

The CSD-principle in PWUL:

```
defprinciple "principle.csdPW" {
  dims {D}
  constraints {

    forall V::node:
      forall V1::node:
        edge(V V1 n D) =>
          forall V2::node:
            forall V3::node:
              dom(V2 V D) & edge(V2 V3 n D) => V3<V1

  }
}
```

PW: Example CSD (Cont'd)

Parsersoutput für: $V2 \rightarrow_D V \wedge V2 \overset{n}{\rightarrow}_D V3$:

```
conj(value(coord:7#7
           sem:dom(value(coord:7
                         sem:constant(token(coord:7
                                            sem:'V2')))
                   value(coord:7
                         sem:constant(token(coord:7
                                            sem:'V')))
                   value(coord:7
                         sem:constant(token(coord:7
                                            sem:'D')))))
     value(coord:7#7
           sem:ledge(value(coord:7
                           sem:constant(token(coord:7
                                              sem:'V2')))
                     value(coord:7
                           sem:constant(token(coord:7
                                              sem:'V3')))
                     value(coord:7
                           sem:constant(token(coord:7
                                              sem:n)))
                     value(coord:7
                           sem:constant(token(coord:7
                                              sem:'D'))))))
```

PW: Type checking/Type inference

- PW contains a Type checker which can also infer missing types.
- Nearly no type annotations are needed any more.
- Works on the parse tree, and extends him with type annotations.

PW: Type checking/Type inference

- Type checker works from the root to the leaves and back.
- On its way, it remeber Variables and corresponding types.
- Written down as rules like:

$$\frac{\Gamma \vdash \texttt{x} :: \texttt{type}}{\Gamma \cup \{x \mapsto type\} \vdash \texttt{x} :: \texttt{type}} x \mapsto T \in \Gamma \Rightarrow T = type$$

PW: Type checking/Type inference

Functions of the Type checker

- CollectDefTypes: Builds a record of types, defined by the user
- UnionContext: Build new context of two existing ones and finds type clashes.

PW: Type checking/Type inference

$\exists$, getypt:
$$\frac{\Gamma \vdash \texttt{exists X :: type : Form} \quad \Gamma \cup \{X \mapsto type\} \vdash \texttt{Form}}{\Gamma \vdash \texttt{exists X::type:Form}} (type \in Dom)$$

$\exists$, ungetypt, inferrierbar:
$$\frac{\Gamma \vdash \texttt{exists X : Form} \quad \Gamma \cup \{X \mapsto type\} \vdash \texttt{Form}}{\Gamma \vdash \texttt{exists X::type:Form}} (type \in Dom)$$

$\exists$, ungetypt, nicht inferrierbar:
$$\frac{\Gamma \vdash \texttt{exists X : Form} \quad \Gamma \vdash \texttt{Form}}{\texttt{Error:CouldnotinferTypeofX}} \begin{array}{l} (con \mapsto T \notin \Gamma) \vee \\ (T = \_) \end{array}$$

PW: Example CSD

The CSD-principle in FOL:

$$csd_d = \forall v, v' :$$
$$v \xrightarrow{n}_d v' \Rightarrow \forall v'', v''' : v'' \rightarrow_d^+ v \wedge v'' \xrightarrow{n}_d v''' \Rightarrow v''' < v'$$

The CSD-principle in PWUL:

```
defprinciple "principle.csdPW" {
  dims {D}
  constraints {

    forall V::node: forall V1::node:
      edge(V V1 n D) =>
        forall V2::node:
          forall V3::node: dom(V2 V D) & edge(V2 V3 n D) => V3<V1
  }
}
```

PW: Example CSD

The CSD-principle in FOL:

$$csd_d = \forall v, v' :$$
$$v \xrightarrow{n}_d v' \Rightarrow \forall v'', v''' : v'' \rightarrow_d^+ v \wedge v'' \xrightarrow{n}_d v''' \Rightarrow v''' < v'$$

The CSD-principle in PWUL (without annotations):

```
defprinciple "principle.csdPW" {
  dims {D}
  constraints {

    forall V: forall V1:
      edge(V V1 n D) =>
        forall V2:
          forall V3: dom(V2 V D) & edge(V2 V3 n D) => V3<V1
  }
}
```

Evaluator

- Generates Mozart/Oz Code, Name, Dimension variables.
- Builds principle definition and node-constraint functors.
- Generated Code uses reifized propagators from Mozart/Oz.

Evaluator: Semantic

Interpretationfunction evaluates rules recusrsive in the Universe (V,M,T), where

- V: Record containing variable names and corresponding names for the Mozart/Oz code, for the different modes.
- M: Indicates which mode must be used (atom, integer, node).
- T: Type assumption.

Evaluator: Semantic

$[\![exists\ Constant :: Dom : Form]\!]^{V,\ M,\ T}$ =

```
if Dom == node then
  NodeRecV = Constant#'NodeRec'
in
  '{PW.existsNodes NodeRecs
    fun {$ '#NodeRecV#'}'#
      [[Form]](V.n ∪ {Constant : NodeRecV}),M,T #
    end}'
else
  [...]
end
```

Evaluator: Example CSD

```
{PW.forallNodes NodeRecs
 fun {$ VNodeRec}
    {PW.forallNodes NodeRecs
     fun {$ V1NodeRec}
        {PW.forallDom {PW.t2Lat label('D')}
         fun {$ LA LI}
            {PW.forallDom {PW.t2Lat label('D')}
             fun {$ L1A L1I}
                {PW.impl
                 {PW.conj
                  {PW.ldom
                     ...
```

Evaluator: Example CSD (Cont'd)

Principle definition:

```
defprinciple "principle.disjPW" {
  dims {D}
  constraints {"DisjPW": 140}}
```

Node-constraint-functor:

```
functor
import
   PW at 'PW.ozf'
export
   Constraint
define
   proc {Constraint NodeRecs G Principle FD FS Select}
      [Constraint]
   end
end
```

Optimisation

- Nested Quantors raise the runtime.
- Experts find out, that the Nodesets of the model can be used to eliminate quantors.
- Some of those techniques could be expressed by patterns.
- The optimiser uses patternmatching to replace patterns in the parsetree with special subtrees.
- Interpretationfunktion generates special code for it.

Optimisation: Example 0or1Mother

Optimisation of the ZeroOrOneMother-principle:

FOL: $\forall v : \underbrace{(\neg\exists v' : v' \rightarrow_d v)}_{|mothers(v)| = 0} \vee \underbrace{(\exists^1 v' : v' \rightarrow_d v)}_{|mothers(v)| = 1}$

Optimiert: $\forall v : \ |mothers(v)| \ \leq \ 1$

Optimisation: Analysis

	Nut1.ul	Diss.ul
Optimised by hand	94 109	1.46 0.655
PWUL-not-optimised	375 468	17 5.65
PWUL-Optimised	172 234	10.21 2.57

Conclusion

- Grammars con now be written analog to the formalisation
- XDK is now more attractive.
- Automated generated principles are efficient enough for practical use,
- especially for rapid prototyping.

Future Work

- Automatic translation of the Formulas to a normalform.
- Further optimisations to eliminate quantors systematically.

Literatur

- Ralph Debusmann (2006). Extensible Dependency Grammar: A Modular Grammar Formalism Based On Multigraph Description. PhD thesis (revised version)
- Ralph Debusmann (2007). Scrambling as the Intersection of Relaxed Context-Free Grammars in a Model-Theoretic Grammar Formalism. ESSLLI 2007 Workshop Model Theoretic Syntax at 10, Dublin
- Mozart Consortium (2006). The Mozart-Oz Website. http://www.mozart-oz.org