A Principle Compiler for Extensible Dependency Grammar

Bachelor Thesis
Programming Systems Lab

Jochen Setz, 08.11.2007

Betreuer: Ralph Debusmann

Introduction

- Extensible Dependency Grammar (XDG) is a constraint-based meta grammar formalism
- Models are formalised by Constraints in First-Order Logic $\Rightarrow$ Principles
- XDG Development Kit (XDK) is the implementation of XDG
- Principles in XDK are Mozart/Oz-Constaints on finite sets

$\Rightarrow$ Implementing new principles is not trivial.

Bachelor's Thesis

- Develop a program (PrincipleWriter) which translates FOL-Constraints into Mozart/Oz-Constraints.
- Close the gap between the formalisation and the implementation of XDG.
- Enable more users to write new principles.
- Speed up grammar writing.
- Increase the attractivity of the development system.

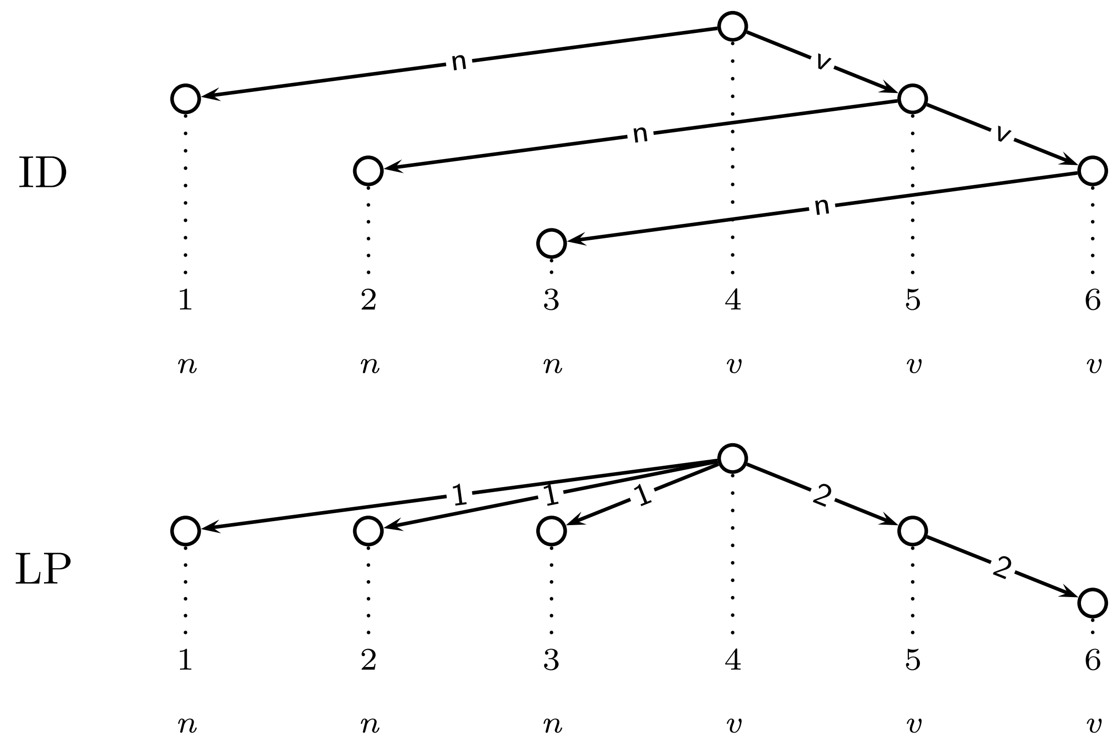
Contents

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

XDG

- XDG is based on dependency grammar.
- XDG grammars are extensible by arbitrary linguistic aspects like word order or predicate-argument structure.
- Each aspect modeled on its own dimension.
- The models of a grammar are represented by multigraphs.
- More modular grammars development

XDG: Multigraphs (Cont'd)



Each node additionally associated with attributes.

XDG: Grammars

A grammar in XDG consists of

- a multigraph type (all possible dimensions, words, edge labels and attributes)
- a lexicon
- 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.
- Similar to copy language (formal grammar).

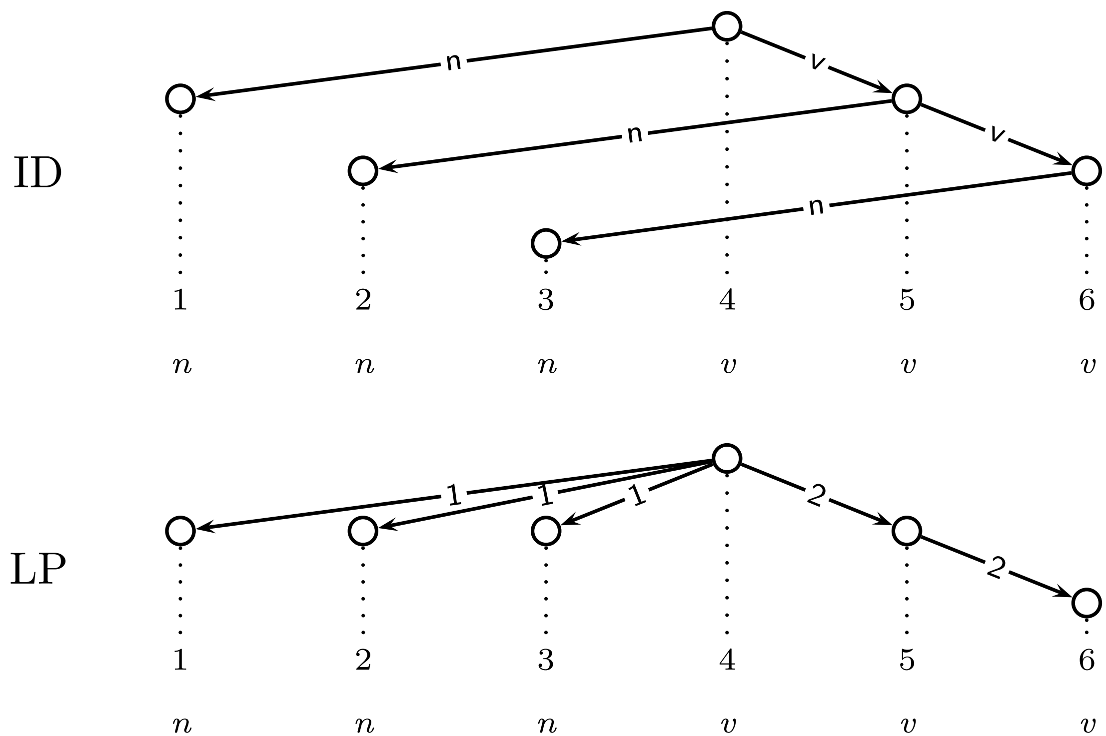
(omdat)	*ik*	*Cecilia*	*Henk*	*de nijlpaarden*	*zag*	*helpen*	*voeren*
(that)	*I*	*Cecilia*	*Henk*	*the hippos*	*saw*	*help*	*feed*

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

XDG: Example CSD (Cont'd)

- Multigraphtype:
  - Dimensions ID (Immediate Dominance) and LP (Linear Precedence).
  - Words n and v
  - Labels: n,v (ID) and 1,2 (LP)
  - Attributes: in, out, order (only LP)

XDG: Example CSD (Cont'd)

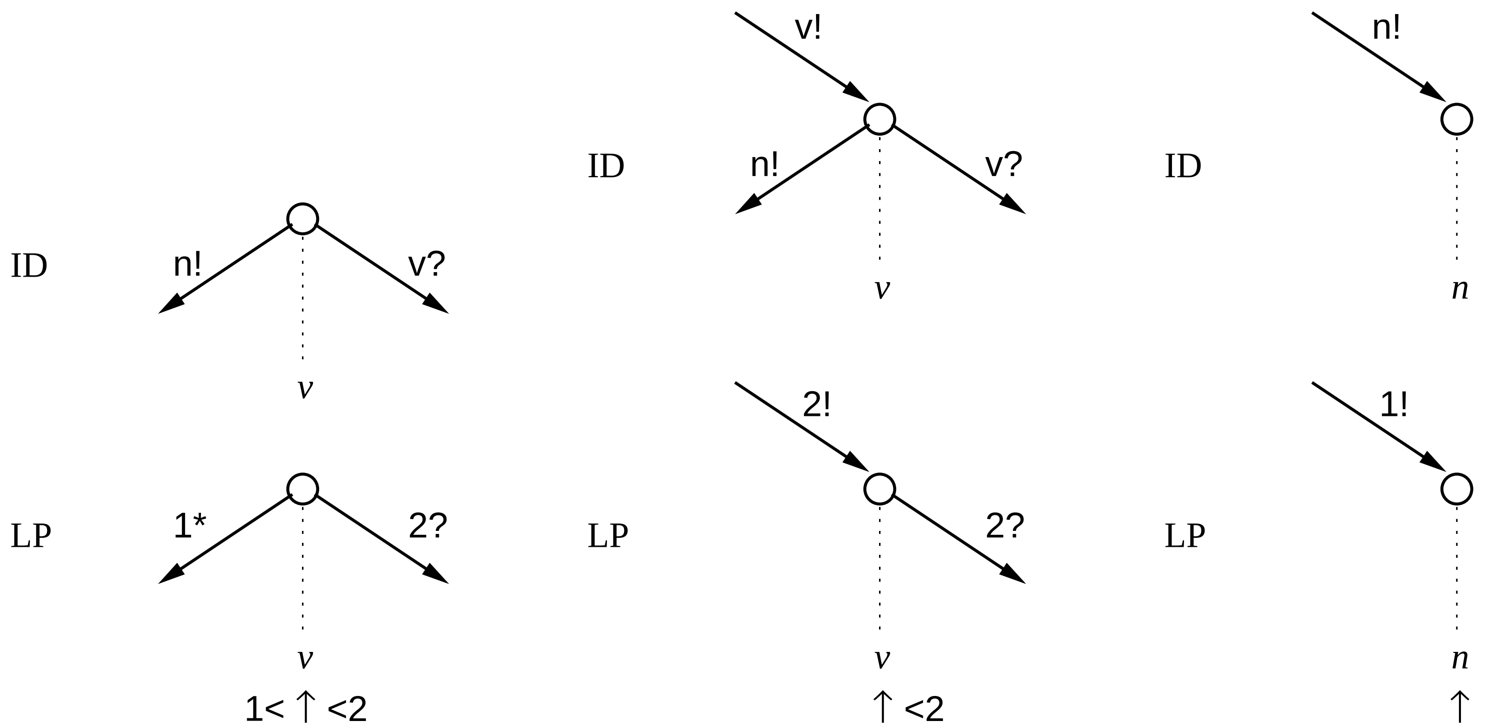


XDG: Lexicon

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

XDG: Lexicon

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



XDG: Principles

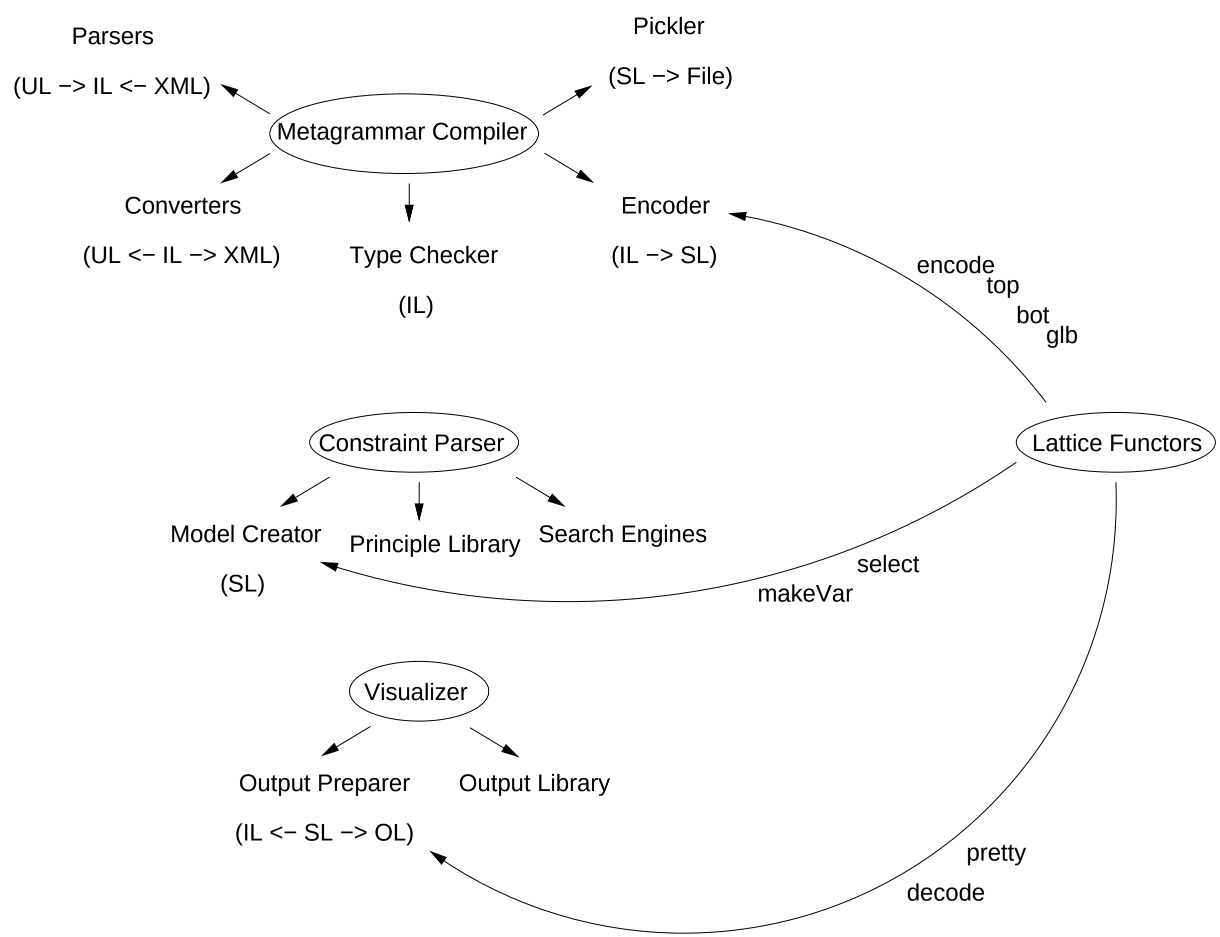
- State the well-formedness conditions of the multigraph.
- Principles are FOL constraints which abstract over dimensions.
- e.g. Tree principle (one root, zero or one mother, no cycles, disjunct subtrees).

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 the n-dependents of the verbs above 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 multigraph type.
- Describes the lexicon.
- The XDK DL cannot describe the principles! DL can just select existing ones from the principle library.
- New principles have to be implemented as constraints on finite sets in Mozart/Oz.

XDK: Description Language

- Describes the multigraph type.
- Describes the lexicon.
- The XDK DL cannot describe the principles! DL can just select existing ones from the principle library.
- New principles have to be implemented as constraints on finite sets in Mozart/Oz.

Grammatik G = (MT, lex, P)

$\uparrow$ ✔ (MT) $\uparrow$ ✔ (lex) $\uparrow$ ✗ (P)

XDK: Multigraphtyp (CSD)

- Mulitgraph type:

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

  [ ... ]
}
```

XDK: Lexicon (CSD)

- The lexicon can be structured using lexical classes.
- Lexical classes are arranged in an inheritance hierarchy.
- Metagrammar compiler: compiles out the class hierarchy into full lexical entries.

XDK: Lexicon (CSD)

- The lexicon can be structured using lexical classes.
- Lexical classes are arranged in an inheritance hierarchy.
- Metagrammar compiler: compiles out the class hierarchy into full lexical entries.
- Example: The verbs in CSD:

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

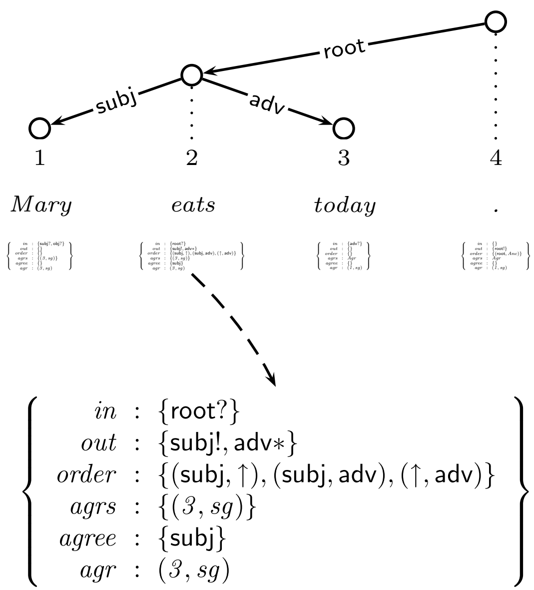
```
defentry {
  "verb" {Word: "v"}
  dim lp {out: {"1"*}}}
```

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

XDK: Principles

- Multigraphs are modeled with finite sets of integers.
- Nodes represented as a Mozart/Oz record with sets variables.
- Principles: Mozart/Oz constraints on the set variables.

XDX: Mulitgraph




```
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 (Cont'd)

- A principle consists of two parts: the principle definition and a set of constraint functors.
- Principle definition specifies abstracted dimension variables and used constraint functors.
- Example: CSD principle:

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

XDK: Principles (Cont'd)

- Example: CSD principle:

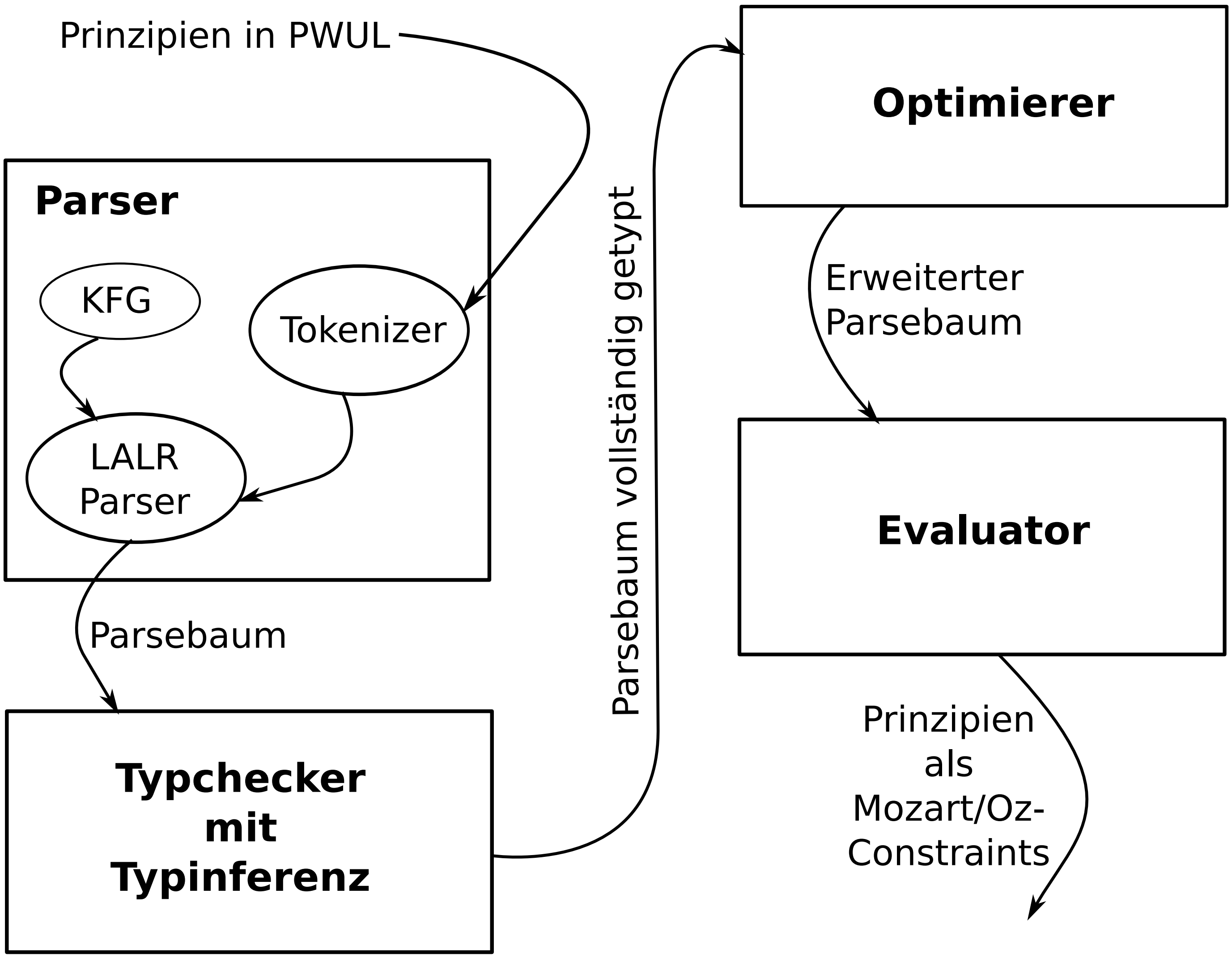
```
proc {Constraint Nodes G GetDim}
   DIDA = {GetDim 'D'}
   PosMs = {Map Nodes
            fun {$ Node} Node.pos end}
in
   for Node in Nodes do
      NDaughtersMs =
      {Map Nodes
       fun {$ Node} Node.DIDA.model.daughtersL.n end}
      NDaughtersUpM = {Select.union NDaughtersMs Node.DIDA.model.up}
      PosNDaughtersUpM = {Select.union PosMs NDaughtersUpM}
      NDaughtersM = Node.DIDA.model.daughtersL.n
      PosNDaughtersM = {Select.union PosMs NDaughtersM}
   in
      {FS.int.seq [PosNDaughtersUpM PosNDaughtersM]}
   end
end
```

- Far away from the formalisation in FOL!

PrincipleWriter

- Formalisation (XDG) $\Leftrightarrow$ Implementation (XDK)
- Multigraph type 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.
  - Especially for the optimisation.
- With the PrincipleWriter, principles can now be automatically translated into Mozart/Oz constraints.

PW: Architecture



PW: User Language

XDG	PWUL	XDG	PWUL	XDG	PWUL
$\neg$	~	$=$	=	$V1 \rightarrow_D V2$	edge(V1 V2 D)
$\wedge$	&	$\neq$	~	$V1 \xrightarrow{L}_D V2$	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	$V1 \xrightarrow{L}{}_D^+ V2$	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)

Parser output 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.
- 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 remembers variables and corresponding types.
- Written down as rules like:

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

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.
- Builds principle definition and constraint functors.
- Generated code uses reified propagators.

Evaluator: Semantics

Interpretation with respect to (V,M,T):

- V: Record mapping variable names to corresponding names in the Mozart/Oz code.
- M: Indicates which mode of encoding must be used (atom, integer, node).
- T: Type assumption.

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 increase the runtime.
- Experts: The additional sets of the model can be used to eliminate quantors.
- Some of these techniques can be expressed by pattern matching.
- The optimiser uses pattern matching to replace patterns in the parsetree with special subtrees.
- Evaluator generates optimised code for them.

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	0.94	1.46
	0.109	0.655
PWUL-not-optimised	0.375	17
	0.468	5.65
PWUL-Optimised	0.172	10.21
	0.234	2.57

Conclusion

- Grammars can now be written analogously to the formalisation
- XDK especially as a meta grammar formalism is now more attractive.
- Automatically generated principles are efficient enough for rapid prototyping.
- Possible to mix handwritten principles from the library with automatically generated ones.

Future Work

- Automatic translation of the formulas to a normal form.
- Find more patterns.
- Further optimisations, e.g. 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 (2007). The Mozart-Oz Website. http://www.mozart-oz.org

Evaluator: Semantics

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

if Dom == node then

NodeRecV = Constant#`'NodeRec'`

in

```
'{PW.existsNodes NodeRecs
  fun {$ '#NodeRecV#'}'#
```

$[\![Form]\!]^{\left(V.n \cup \{Constant : NodeRecV\} \right),M,T}$ #

```
  end}'
```

else

[...]

end