11

Modular Grammar Design with Typed Parametric Principles

Ralph Debusmann, Denys Duchier and Andreas Rossberg

Abstract

This paper introduces a type system for *Extensible Dependency Grammar* (XDG) (Debusmann et al., 2004), a new, modular grammar formalism based on dependency grammar. As XDG is based on graph description, our emphasis is on capturing the notion of *multigraph*, a tuple of arbitrary many graphs sharing the same set of nodes. An XDG grammar consists of the stipulation of an extensible set of *parametric principles*, which yields a modular and compositional approach to grammar design.

Keywords grammar formalisms, dependency grammar, type systems

11.1 Introduction

Extensible Dependency Grammar (XDG) (Debusmann et al., 2004) is a general framework for dependency grammar, with multiple levels of linguistic representations called *dimensions*. Its approach, motivated by the dependency grammar paradigm (Tesnière, 1959, Mel'čuk, 1988), is articulated around a description language for multi-dimensional attributed labeled graphs. XDG is a generalization of *Topological Dependency Grammar* (TDG) (Duchier and Debusmann, 2001).

For XDG, a grammar is a constraint that describes the valid linguistic signs as n-dimensional attributed labeled graphs, i.e. n-tuples of graphs sharing the same set of attributed nodes, but having different sets of labeled edges. It is central to XDG that all aspects of these signs are

FG-MoL 2005.
James Rogers (ed.).

stipulated explicitly by *principles*: the class of models for each dimension, additional properties that they must satisfy, how one dimension must relate to another, and even lexicalization.

Yet, no formal account set in the XDG framework has so far explained what exactly these principles are, nor how they can be brought to bear on specific dimensions. In this paper, we show how an XDG grammar can be formally assembled from modular components called *parametric principles*. This yields a modular and compositional approach to grammar design. Compositional coherence is ensured by a *type system* whose primary novelty is to accommodate the notion of multi-dimensional graphs. Instantiation of parametric principles not only imposes grammatical constraints, but, through the type system, also determines the necessary structure of grammatical signs. In this perspective, a grammar framework is simply a library of parametric principles such as the one offered by the XDG *Development Kit* (XDK) (Debusmann and Duchier, 2004). The XDK is a freely available development environment for XDG grammars including a concurrent constraint parser written in the Mozart/Oz programming language (Mozart Consortium, 2005).

11.2 Extensible Dependency Grammar

We briefly illustrate the XDG approach with an example of the German subordinate sentence *"(dass) einen Mann Maria zu lieben versucht"*.[1] Figure 1 shows an analysis with two dimensions: ID models grammatical function and LP word-order using *topological fields* (Duchier and Debusmann, 2001, Gerdes and Kahane, 2001)

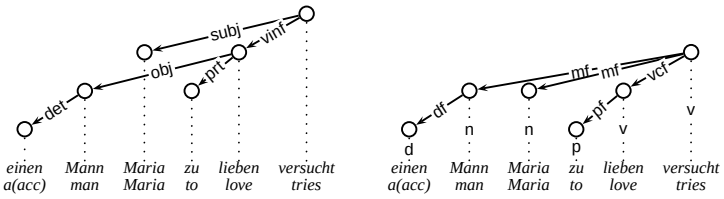

FIGURE 1: Example XDG analysis, ID left, LP right

Both dimensions share the same set of nodes (circles) but have different edges. On ID, *Maria* is subject (subj) of control verb *versucht*, and *Mann* object (obj) of *lieben*. On LP, *Mann* and *Maria* are both in the *Mittelfeld* (mf) of *versucht*, i.e. *Mann* has *climbed* to the finite verb.

[1] (that) Mary tries to love a man – see (Duchier and Debusmann, 2001)

The instance of XDG demonstrated above covers only syntax. However, XDG instances can use arbitrary many dimensions of representation, e.g. e.g. including semantics (including predicate-argument structure and scope relationships) (Debusmann et al., 2004), prosody and information structure (Debusmann et al., 2005).

In fact, XDG per se is not a grammar formalism but only a general graph description language for multi-dimensional attributed labeled graphs. In order to be used as a grammar formalism, it first needs to be instantiated using appropriate principles (see below). In this respect, it is similar to *Head-driven Phrase Structure Grammar* (*HPSG*) (Pollard and Sag, 1994), which per se is only a description language for typed feature structures (Carpenter, 1992), and becomes a grammar formalism only when it is instantiated, using appropriate types for the feature structures and the principles for e.g. head feature percolation and subcategorization.

In particular, XDG is more general than other dependency-based grammar formalisms like *Meaning Text Theory* (*MTT*) (Mel'čuk, 1988), *Functional Generative Description* (*FGD*) (Sgall et al., 1986), *Word Grammar* (*WG*) (Hudson, 1990) and *Free Order Dependency Grammar* (*FODG*) (Holan et al., 2000) since it can accommodate arbitrary many dimensions of representation and arbitrary principles to stipulate the well-formedness conditions. For instance, the dimensions of XDG need not be trees but can also be directed acyclic graphs (dags), e.g. to handle re-entrancies for the modeling of control constructions or relative clauses. In addition, the dimensions can be totally agnostic to word order. In our example above, the ID dimension did not state any word order requirements, only the LP dimension did.

11.3 Type System

Formalization. Let $\mathcal{V}$ be an infinite set of node variables, $\mathcal{L}$ a finite set of labels, and $\mathcal{A}$ a set of attributes. An *attributed labeled graph* (V, E, A) consists of a finite set of nodes $V \subseteq \mathcal{V}$, a finite set of labeled directed edges $E \subseteq V \times V \times \mathcal{L}$ between them, and an assignment $A : V \to \mathcal{A}$ of attributes to nodes. An *n-dimensional attributed labeled graph* $((V, E_1, A_1), \ldots, (V, E_n, A_n))$, or *multigraph*, is a n-tuple of labeled attributed graphs (V, E_i, A_i) over the same set of nodes V.

To provide a typed account of the XDG framework, we need a satisfactory type for graphs. A first idea is $(\{\mathcal{V}\}, \{\mathcal{V} \times \mathcal{V} \times \mathcal{L}\}, \mathcal{V} \to \mathcal{A})$, where we write $\{\tau\}$ for *set of* τ, but such a type is very imprecise: it fails to express that the nodes used in the edges (2nd arg) are elements of the graph's set of nodes (1st arg). Since additionally element

graphs of a multigraph must be defined over the *same* set of nodes, some form of dependent typing (Aspinall and Hofmann, 2005) appears inescapable. Given that general systems of dependent types tend to make type-checking undecidable, we sketch instead a stratified system, using a specialised notion of kinds (Pierce, 2002), that is sufficient for our purpose.

11.3.1 Type structure

We assume given a number of disjoint sets of symbols D_i called finite domain kinds. Among them, two are distinguished: Nodes and Labels. Much of our type and kinding systems are quite standard. For reasons of space, we only detail the parts which are original to our proposal, and write $\tau :: \kappa$ and $e : \tau$ for the judgments that type τ has kind κ and expression e has type τ, omitting kinding and typing contexts.

$$\begin{array}{rlr} \kappa ::= & \star & \text{top} \\ | & \mathsf{D}_i & \text{domain} \\ | & \mathsf{G}_{c_1|\cdots|c_k} & \text{graph} \\ | & \mathsf{M}_{c_1|\cdots|c_k} & \text{multigraph} \\ | & \ldots & \end{array} \qquad \mathsf{D}_i \sqsubset \star \quad \mathsf{G}_v \sqsubset \star \quad \mathsf{M}_v \sqsubset \star \qquad \frac{\tau :: \kappa \quad \kappa \sqsubset \kappa'}{\tau :: \kappa'}$$

FIGURE 2: Kinds and subkinds

$$\begin{array}{rlll|rlll} \tau, \tau_i ::= & c_1|\cdots|c_k & \text{domain} & | & [f_1 : \tau_1, \ldots, f_n : \tau_n] & \text{record} \\ | & \{\tau\} & \text{set} & | & \mathsf{graph}\ \tau_1\ \tau_2\ \tau_3 & \text{graph} \\ | & (\tau_1, \ldots, \tau_n) & \text{tuple} & | & [\![f_1 : \tau_1, \ldots, f_n : \tau_n]\!] & \text{multigraph} \\ | & \tau_1 \to \tau_2 & \text{function} & | & \mathsf{grammar}\ \tau & \text{grammar} \\ | & !\tau & \text{singleton} & | & \ldots & \end{array}$$

FIGURE 3: Types

Finite domain sum types. They are built from symbols drawn from finite domain kinds: we write $c_1|\cdots|c_k$ for a finite domain sum type in kind D_i and ϵ_{D_i} for its empty sum. Here are their kinding (left) and typing (right) rules:

$$\frac{c_1, \ldots, c_k \text{ symbols in } \mathsf{D}_i}{c_1|\cdots|c_k :: \mathsf{D}_i} \qquad \frac{c_1|\cdots|c_k :: \mathsf{D}_i}{c_i : c_1|\cdots|c_k}$$

Kinds $\mathsf{G}_{c_1|\cdots|c_k}$ and $\mathsf{M}_{c_1|\cdots|c_k}$ indexed by finite domain sums make it possible to have types that depend on specific subsets of Nodes or Labels

without requiring more general term-dependent types.

Singleton types. If $c_1|\cdots|c_k :: \mathsf{Nodes}$ is the type of nodes of a graph, then $\{c_1|\cdots|c_k\}$ is the type of a set of these nodes. This is not sufficiently precise to type the set of nodes of the graph because it doesn't express that the latter is a maximal set. To achieve this aim, we introduce a novel, specialised variation of *singleton types* (Aspinall, 1995, Stone and Harper, 2005). Unlike standard singleton types, our form does not refer to values, hence we avoid the need for dependent types. We write $!(c_1|\cdots|c_k)$ for the type inhabited by the single value $\{c_1,\ldots,c_k\}$. Here are the relevant kinding, typing, and subtyping rules:

$$\frac{\tau :: \mathsf{D}_i}{!\tau :: \star} \qquad \frac{!(c_1|\cdots|c_k) :: \star}{\{c_1,\ldots,c_k\} : !(c_1|\cdots|c_k)} \qquad !\tau \sqsubseteq \{\tau\}$$

Graph types. A graph is defined from finite domain types ν and ℓ for its nodes and labels, and a type a for its attributes. We write G_ν for the kind of a graph over node type ν. Kinding and typing rules are:

$$\frac{\nu :: \mathsf{Nodes} \quad \ell :: \mathsf{Labels} \quad a :: \star}{\mathsf{graph}\ \nu\ \ell\ a :: \mathsf{G}_\nu} \qquad \frac{V : !\nu \quad E : \{(\nu,\nu,\ell)\} \quad A : \nu \to a}{\mathsf{Graph}\ V\ E\ A : \mathsf{graph}\ \nu\ \ell\ a}$$

The typing rule requires the node set to be assigned a singleton type, capturing the precise set of nodes in the type ν. Typing thus precludes the set of edges mentioning invalid nodes. Note also that the syntax of graph kinds in Figure 3 requires ν to be a concrete domain type.

Multigraph types. A multigraph is a record of graphs over the same finite domain ν of nodes. We write M_ν for its kind. Here are the kinding and typing rules:

$$\frac{g_1 :: \mathsf{G}_\nu \quad \ldots \quad g_n :: \mathsf{G}_\nu}{[\![f_1 : g_1,\ldots,f_n : g_n]\!] :: \mathsf{M}_\nu}$$

$$\frac{G_1 : g_1 :: \mathsf{G}_\nu \quad \ldots \quad G_n : g_n :: \mathsf{G}_\nu}{[\![f_1 = G_1,\ldots,f_n = G_n]\!] : [\![f_1 : g_1,\ldots,f_n : g_n]\!] :: \mathsf{M}_\nu}$$

Grammars. An XDG *grammar* is a set of predicates over the same multigraph type:

$$\frac{\tau :: \mathsf{M}_\nu}{\mathsf{grammar}\ \tau :: \star} \qquad \frac{\tau :: \mathsf{M}_\nu \quad S : \{\tau \to \mathsf{prop}\}}{\mathsf{Grammar}\ S : \mathsf{grammar}\ \tau}$$

Note that, in order to match our intuitions about grammars, the grammar type should be polymorphic in the finite domain type for nodes, otherwise the number of nodes is fixed. This can be achieved by extending the kinding system to admit *kind schemes* G_δ and M_δ where δ is a domain variable. An XDG *framework* is a set, also called a library, of principle templates.

11.4 Typed Templates

Attributes are usually given in the form of attribute/value matrices and principles parametrized by values which can be found at specific feature paths. For example, on the syntactic dimension $(V, E_{\text{ID}}, A_{\text{ID}})$ the agreement tuple assigned to each word must be one of those licensed by its lexical entry:[2]

$$\text{Agr}\ [\![\mathsf{id} = \mathsf{Graph}\ V\ E_{\text{ID}}\ A_{\text{ID}}]\!] = \quad \forall v \in V : A_{\text{ID}}(v).\mathsf{agr} \in A_{\text{ID}}(v).\mathsf{lex.agrs}$$

We can generalize this into a reusable principle by abstracting over feature paths using access functions:

$$\begin{array}{rcl} \text{Elem}\ D\ F_1\ F_2\ M & = & \textbf{let}\ \mathsf{Graph}\ V\ E\ A = D(M) \\ & & \textbf{in}\ \forall v \in V : F_1(A(v)) \in F_2(A(v)) \end{array}$$

but this is not very legible and, for notational convenience, we explore here an alternative that we call *templates*:

$$\text{Elem}\langle d, p_1, p_2\rangle\ [\![d = \mathsf{Graph}\ V\ E\ A]\!] \quad = \quad \forall v \in V : A(v).p_1 \in A(v).p_2$$

where d, p_1, p_2 are *feature path variables*. A feature path is a (possibly empty) sequence of features. We write π for a path, ϵ for the empty path, and $\pi_1\pi_2$ for the concatenation of π_1 and π_2. It is possible to generalize the language by allowing feature paths or feature path variables in types and patterns (and by extension in record 'dot' access) where previously only features were allowed. The intuition of such an extension lies in the congruence $[\epsilon : \tau] \equiv \tau$, $[\pi_1\pi_2 : \tau] \equiv [\pi_1 : [\pi_2 : \tau]]$, and in the interpretation of a dot access $.\pi$ as a postfix function of type $[\pi : \tau] \to \tau$.

Note that, if we write $\mathsf{graph}\ \nu\ \ell\ a$ for the type of Elem's argument graph, it is our intention that type inference should require a to match the pattern $[p_1 : \tau,\ p_2 : \{\tau\}, \ldots]$. This can be achieved either with a type system supporting record polymorphism or by adopting an open-world semantics[3] for records and multigraphs, à la ψ-terms of LIFE. For simplicity, in this article we choose the latter.

We write $\langle p_1, p_2\rangle \rightsquigarrow \tau$ for the type of a template abstracting over feature path variables p_1 and p_2; t may contain occurrences of p_1 and p_2. We write $\tau_1 :: \kappa_1, \ldots, \tau_n :: \kappa_n \Rightarrow \tau$ to express kinding constraints on the free type variables of τ. The type of Elem is then:

$$\begin{array}{r} \text{Elem} : \langle d, p_1, p_2\rangle \rightsquigarrow \nu :: \mathsf{Nodes},\ \ell :: \mathsf{Labels},\ \tau :: \star \Rightarrow \\ [\![d : \mathsf{graph}\ \nu\ \ell\ [p_1 : \tau,\ p_2 : \{\tau\}]]\!] \to \mathsf{prop} \end{array}$$

[2] We adopt a pattern matching notation with obvious meaning

[3] no closed arities

11.5 Parametric Principles

We now illustrate how typed templates are intended to be used to define *parametric principles*.[4] We write $v \xrightarrow{l}_E v'$ for an edge $(v, v', l) \in E$, $v \rightarrow_E v'$ for one with any label, $v \rightarrow_E^+ v'$ for the transitive closure, and $v \rightarrow_E D$ for the type-raised relation where $D = \{v' \mid \forall v \rightarrow_E v'\}$, etc...

Tree Principle. It stipulates the set L of edge labels and requires that 1) each node has at most one incoming edge, 2) there is precisely one node with no incoming edge (one root), and 3) there are no cycles:

$$\begin{array}{l}\mathrm{Tree}\langle d\rangle\ L\ [\![d = \mathsf{Graph}\ V\ E\ A]\!] = \\ \quad \forall v \in V \quad : \quad \forall M \subseteq V : M \rightarrow_E v \Rightarrow |M| \leq 1 \quad \wedge \\ \quad \exists! v \in V \quad : \quad \forall M \subseteq V : M \rightarrow_E v \Rightarrow |M| = 0 \quad \wedge \\ \quad \forall v \in V \quad : \quad \forall D \subseteq V : v \rightarrow_E^+ D \Rightarrow v \notin D\end{array}$$

Here is the type constraint assigning singleton type $!\ell$ to L:

$$\mathrm{Tree} : \langle d\rangle \rightsquigarrow \nu :: \mathsf{Nodes}, \ell :: \mathsf{Labels} \Rightarrow\ !\ell \rightarrow [\![d : \mathsf{graph}\ \nu\ \ell\ \_]\!] \rightarrow \mathsf{prop}$$

Valency Principle. Incoming and outgoing edges must comply with the *in* and *out* valencies which stipulate, for each label $\ell \in L$, how many edges labeled with ℓ are licensed:

$$\begin{array}{l}\mathrm{Valency}\langle d, p_{\mathsf{in}}, p_{\mathsf{out}}\rangle\ L\ [\![d = \mathsf{Graph}\ V\ E\ A]\!] = \\ \quad \forall v \in V\, \forall \ell \in L \quad : \quad \forall M \subseteq V : M \xrightarrow{\ell}_E v \Rightarrow |M| \in A(v).p_{\mathsf{in}}.\ell \quad \wedge \\ \quad \forall v \in V\, \forall \ell \in L \quad : \quad \forall D \subseteq V : v \xrightarrow{\ell}_E D \Rightarrow |D| \in A(v).p_{\mathsf{out}}.\ell\end{array}$$

Writing $\mathbb{N}$ for the type of natural numbers, here is the type constraint:

$$\begin{array}{l}\mathrm{Valency} : \langle d, p_{\mathsf{in}}, p_{\mathsf{out}}\rangle \rightsquigarrow \nu :: \mathsf{Nodes}, \ell :: \mathsf{Labels} \Rightarrow \\ !\ell \rightarrow [\![d : \mathsf{graph}\ \nu\ \ell\ [p_{\mathsf{in}} : \ell \rightarrow \{\mathbb{N}\}, p_{\mathsf{out}} : \ell \rightarrow \{\mathbb{N}\}]]\!] \rightarrow \mathsf{prop}\end{array}$$

Climbing Principle. Originating from TDG, this principle expresses that the tree-shape on dimension d_1 is a flattening of that of d_2: 1) the dominance relation on d_1 must be a subset of that on d_2, 2) on d_1, each node must land on its d_2-mother or climb higher:

$$\begin{array}{l}\mathrm{Climbing}\langle d_1, d_2\rangle [\![d_1 = \mathsf{Graph}\ V\ E_1\ A_1, d_2 = \mathsf{Graph}\ V\ E_2\ A_2]\!] = \\ \quad \forall v \in V \quad : \quad \forall D_1, D_2 \subseteq V : v \rightarrow_{E_1}^+ D_1 \wedge v \rightarrow_{E_2}^+ D_2 \Rightarrow D_1 \subseteq D_2 \quad \wedge \\ \qquad\qquad\quad \forall U_1, U_2 \subseteq V : U_1 \rightarrow_{E_1}^+ v \wedge U_2 \rightarrow_{E_1}^* \rightarrow_{E_2} v \Rightarrow U_1 \subseteq U_2\end{array}$$

$$\begin{array}{l}\mathrm{Climbing} : \langle d_1, d_2\rangle \rightsquigarrow \nu :: \mathsf{Nodes} \Rightarrow \\ [\![d_1 : \mathsf{graph}\ \nu\ \_\ \_, d_2 : \mathsf{graph}\ \nu\ \_\ \_]\!] \rightarrow \mathsf{prop}\end{array}$$

[4] For lack of space, we present only a few principles. For further information, the reader is referred to e.g. (Debusmann et al., 2004, Debusmann and Duchier, 2004).

Lexicon Principle. XDG grammars are typically *lexicalized*: the record assignments to nodes are then partially determined by a *lexicon* Lex. The lexicon principle stipulates that each node must be assigned a *lexical entry*:

$$\text{Lexicon}\langle d\rangle\ Lex\ [\![d = \mathsf{Graph}\ V\ E\ A]\!] = \forall v \in V : A(v) \in Lex$$

Here is the corresponding type constraint, stipulating that the graph on dimension d has no edges (since this dimension has only the purpose to carry the lexical entries):

$$\begin{array}{l}\text{Lexicon} : \langle d\rangle \rightsquigarrow \nu :: \mathsf{Nodes}, \tau :: \star \Rightarrow \\ \{\tau\} \rightarrow [\![d : \mathsf{graph}\ \nu\ \epsilon_{\mathsf{Labels}}\ \tau]\!] \rightarrow \mathsf{prop}\end{array}$$

Lookup Principle. A lexical entry is normally a record having a feature for each dimension. The *lookup* principle looks up a lexical entry's subrecord for a particular dimension and equates it with the p_{lex} feature of the node's attributes:

$$\begin{array}{l}\text{Lookup}\langle d_1, d_2, p_{\mathsf{lex}}\rangle [\![d_1 = \mathsf{Graph}\ V\ E_1\ A_1, d_2 = \mathsf{Graph}\ V\ E_2\ A_2]\!] = \\ \quad \forall v \in V \quad : \quad A_1(v).p_{\mathsf{lex}} = A_2(v).d_1\end{array}$$

$$\begin{array}{l}\text{Lookup} : \langle d_1, d_2, p_{\mathsf{lex}}\rangle \rightsquigarrow \nu :: \mathsf{Nodes}, \tau :: \star \Rightarrow \\ [\![d_1 : \mathsf{graph}\ \nu\ \_\ [p_{\mathsf{lex}} : \tau], d_2 : \mathsf{graph}\ \nu\ \_\ [d_1 : \tau]]\!] \rightarrow \mathsf{prop}\end{array}$$

11.6 Example ID/LP Grammar

We now describe how the grammar of (Duchier and Debusmann, 2001) can be assembled in our typed framework of parametric principles. To better illustrate the compositionality of our approach, we adopt an incremental presentation that derives more complex grammars from simpler ones through operations of composition and restriction:

$$\begin{array}{rcl}(\mathsf{Grammar}\ S_1) +\!\!+ (\mathsf{Grammar}\ S_2) & = & \mathsf{Grammar}\ (S_1 \cup S_2) \\ (\mathsf{Grammar}\ S_1)\ //\ S_2 & = & \mathsf{Grammar}\ (S_1 \cup S_2)\end{array}$$

The grammar requires 3 dimensions: ID for syntax, LP for topology, and LEX for parametrization by a lexicon. They are respectively characterized by the following sets of labels:

$$\begin{array}{rcl}L_{\text{ID}} & = & \{\mathsf{det}, \mathsf{subj}, \mathsf{obj}, \mathsf{vbse}, \mathsf{vprt}, \mathsf{vinf}, \mathsf{prt}\} \\ L_{\text{LP}} & = & \{\mathsf{d}, \mathsf{df}, \mathsf{n}, \mathsf{mf}, \mathsf{vcf}, \mathsf{p}, \mathsf{pf}, \mathsf{v}, \mathsf{vxf}\} \\ L_{\text{LEX}} & = & \emptyset\end{array}$$

Figure 4 shows how grammars for the 3 individual dimensions (G_{ID}, G_{LP}, G_{LEX}) are stipulated by instantiation of principles. For example, for G_{ID}, instantiation of $\text{Tree}\langle\mathsf{id}\rangle\ L_{\text{ID}}$ has three consequences: (1) signs are required to match $[\![\mathsf{id} = X]\!]$, i.e. to have an id dimension, (2) the

directed graph on that dimension must have labels in L_{ID}, (3) this graph must be a tree. $G_{\text{ID+LEX}}$ is a grammar where the ID dimension is restricted by lexical constraints from the LEX dimension: it is obtained by combining G_{ID} and G_{LEX} using the composition operator ++, and connecting them by the Lookup$\langle \mathsf{id}, \mathsf{lex} \rangle$ principle using the restriction operator //. Similarly for $G_{\text{LP+LEX}}$. Finally the full grammar $G_{\text{ID/LP}}$ is obtained by combining $G_{\text{ID+LEX}}$ and $G_{\text{LP+LEX}}$ and mutually constraining them by the Climbing and Barriers principles.

$$
\begin{array}{lcl}
G_{\text{LEX}} & = & \mathsf{Grammar}\ \{\text{Lexicon}\langle \mathsf{lex} \rangle\ \text{Lex}\} \\
G_{\text{ID}} & = & \mathsf{Grammar}\ \{\text{Tree}\langle \mathsf{id} \rangle\ L_{\text{ID}}, \text{Valency}\langle \mathsf{id}, \mathsf{in}, \mathsf{out} \rangle\ L_{\text{ID}}\} \\
G_{\text{ID+LEX}} & = & (G_{\text{ID}} \mathbin{++} G_{\text{LEX}}) \mathbin{//} \{\text{Lookup}\langle \mathsf{id}, \mathsf{lex}, \mathsf{lex} \rangle\} \\
G_{\text{LP}} & = & \mathsf{Grammar}\ \{\text{Tree}\langle \mathsf{lp} \rangle\ L_{\text{LP}}, \text{Valency}\langle \mathsf{lp}, \mathsf{in}, \mathsf{out} \rangle\ L_{\text{LP}}, \\
 & & \qquad \text{Order}\langle \mathsf{lp}, \mathsf{on}, \mathsf{order}, \mathsf{pos}, \mathsf{self} \rangle, \\
 & & \qquad \text{Projectivity}\langle \mathsf{lp}, \mathsf{pos} \rangle\} \\
G_{\text{LP+LEX}} & = & (G_{\text{LP}} \mathbin{++} G_{\text{LEX}}) \mathbin{//} \{\text{Lookup}\langle \mathsf{lp}, \mathsf{lex}, \mathsf{lex} \rangle\} \\
G_{\text{ID/LP}} & = & (G_{\text{ID+LEX}} \mathbin{++} G_{\text{LP+LEX}}) \mathbin{//} \\
 & & \qquad \{\text{Climbing}\langle \mathsf{lp}, \mathsf{id} \rangle, \text{Barriers}\langle \mathsf{lp}, \mathsf{id}, \mathsf{blocks} \rangle\}
\end{array}
$$

FIGURE 4: Defining grammars in an incremental and compositional way

11.7 Conclusion

In this paper, we have made a threefold contribution: (1) we described a novel system of restricted dependent types capable of precisely describing graphs and multi-dimensional graphs, (2) we introduced a notion of typed templates with which we could express parametric principles, (3) we showed how these could enable a modular and compositional approach to grammar design. Finally we illustrated our proposal with an example reconstruction of an earlier grammar.

References

Aspinall, David. 1995. Subtyping with singleton types. In *Computer Science Logic*, vol. 933 of *Lecture Notes in Computer Science*. Springer-Verlag.

Aspinall, David and Martin Hofmann. 2005. Dependent types. In B. Pierce, ed., *Advanced Topics in Types and Programming Languages*, chap. I.2, pages 45–136. The MIT Press.

Carpenter, Bob. 1992. *The Logic of Typed Feature Structures*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 32nd edn.

Debusmann, Ralph and Denys Duchier. 2004. XDG Development Kit. http://www.ps.uni-sb.de/~rade/mogul/publish/doc/debusmann-xdk/.

Debusmann, Ralph, Denys Duchier, Alexander Koller, Marco Kuhlmann, Gert Smolka, and Stefan Thater. 2004. A Relational Syntax-Semantics Interface Based on Dependency Grammar. In *Proceedings of COLING 2004*. Geneva/CH.

Debusmann, Ralph, Oana Postolache, and Maarika Traat. 2005. A Modular Account of Information Structure in Extensible Dependency Grammar. In *Proceedings of the CICLING 2005 Conference*. Mexico City/MEX: Springer.

Duchier, Denys and Ralph Debusmann. 2001. Topological Dependency Trees: A Constraint-Based Account of Linear Precedence. In *Proceedings of ACL 2001*. Toulouse/FR.

Gerdes, Kim and Sylvain Kahane. 2001. Word Order in German: A Formal Dependency Grammar Using a Topological Hierarchy. In *ACL 2001 Proceedings*. Toulouse/FR.

Holan, Tomás, Vladislav Kubon, Karel Oliva, and Martin Platek. 2000. On Complexity of Word-Order. *Journal t.a.l.* pages 273–301.

Hudson, Richard A. 1990. *English Word Grammar*. Oxford/UK: B. Blackwell.

Mel'čuk, Igor. 1988. *Dependency Syntax: Theory and Practice*. Albany/US: State Univ. Press of New York.

Mozart Consortium. 2005. The Mozart-Oz Website. http://www.mozart-oz.org/.

Pierce, Benjamin. 2002. *Types and Programming Languages*. The MIT Press.

Pollard, Carl and Ivan A. Sag. 1994. *Head-Driven Phrase Structure Grammar*. Chicago/US: University of Chicago Press.

Sgall, Petr, Eva Hajicova, and Jarmila Panevova. 1986. *The Meaning of the Sentence in its Semantic and Pragmatic Aspects*. Dordrecht/NL: D. Reidel.

Stone, Christopher and Robert Harper. 2005. Extensional equivalence and singleton types. *Transactions on Computational Logic* to appear.

Tesnière, Lucien. 1959. *Eléments de Syntaxe Structurale*. Paris/FR: Klincksiek.