Configuration Of Labeled Trees
Under Lexicalized Constraints And Principles

Denys Duchier

Programming Systems Lab, Universität des Saarlandes, Saarbrücken,
Email: `duchier@ps.uni-sb.de`

ABSTRACT: Trees with labeled edges have widespread applicability, for example for the representation of dependency syntax trees. Given a fixed number of nodes and constraints on how edges may be drawn between them, the task of finding solution trees is known as a configuration problem. In this paper, we formalize the configuration problem of labeled trees and argue that it can be regarded as a constraint satisfaction problem which can be solved directly and efficiently by constraint propagation. In particular, we derive and prove correct a formulation of dependency parsing as a constraint satisfaction problem.

Our approach, based on constraints on finite sets and a new family of 'selection' constraints, is especially well-suited for the compact representation and efficient processing of ambiguity. We address various issues of interest to the computational linguist such as lexical ambiguity, structural ambiguity, valency constraints, grammatical principles, and linear precedence. Finally we turn to the challenge of efficient processing and characterize the services expected of a constraint programming system: we define a formal constraint language and specify its operational semantics with inference rules of propagation and distribution.

This framework generalizes our presentation of immediate syntactic dependence for dependency parsing [4] and extends naturally to our corresponding treatment of linear precedence [6] based on a notion of topological rather than syntactic dependencies.

KEYWORDS: *labeled trees, configuration, constraint satisfaction, constraint propagation, set constraints, parsing, dependency grammar*

1 Introduction

This article presents a formalization of finite trees with labeled edges: well-formedness is characterized by a small number of equations and trees correspond precisely to the solutions of these equations. The advantage of our approach is that all our equations can be interpreted as constraints and can be solved directly using constraint programming technology. The constraint-based approach is especially well-suited for the compact representation and efficient processing of ambiguity, and constraint propagation is very effective in pruning the search space.

We begin with a characterization of the legal trees which can be assembled from a finite set V of nodes and a finite set $\mathcal{L}$ of edge labels. Arranging the nodes V into a tree requires choosing, from the set of all possible labeled edges, a subset such that the resulting graph is a tree. This task may be regarded as a configuration problem and can be formulated as constraint satisfaction problem well-suited to a constraint-based approach. We then entertain various refinements, especially relevant to linguistics, where admissibility is further restricted either by general principles or through lexicalized constraints. Finally, we address the issue of efficient processing: it is achieved by effective model elimination through constraint propagation. For this reason, we give precise operational semantics to all our constraints in the form of inference rules. The search for solutions of a constraint satisfaction problem (CSP) is defined formally as the derivation of consistent saturations under these inference rules.

The objects described by modern linguistic theories such as HPSG [21, 22] or LFG [15] are typed features structures (TFS) and the theories themselves consist primarily in the formulation of general structural principles of well-formedness that determine which of these objects are licensed. These theories are declarative and constraint-based and remain uncommitted to any particular processing method. While TFS are appealing and easily integrated in unification-based computational frameworks, they result in grammatical formalisms which are hard to process efficiently. The processing challenge is further exacerbated when attempting to account for languages, such as German, where free word order and discontinuous constituents violate the assumptions of linearity and adjacency underlying many parsing techniques.

In [4], we described an alternative approach based on Dependency Grammar (DG). An advantage of DG is that it allows syntax trees with crossing branches, and thus does not fall prey to the difficulties we just mentioned plaguing grammatical formalisms that have traditionally assumed and required projective analyses. We showed how parsing could be formulated succinctly as a constraint satisfaction problem (CSP) solvable efficiently by constraint programming. One novelty of our approach was the central importance given to sets and constraints on set variables.

Our purpose in the present article is twofold. Firstly, we abstract the approach of [4] away from the details specific to dependency grammar. We tackle the more general problem of configuring trees with labeled edges and demonstrate how it can be formulated as a CSP. In this manner, our techniques gain wider scope, for example extending naturally to the treatment of linear precedence described in [6]. Secondly, we give a formal account of constraint programming sufficient to solve the aforementioned CSPs: we define an abstract constraint language, completely specify its operational semantics by means of inference rules for propagation and distribution, and prove that solutions precisely correspond to the consistent saturations. It is our hope that, by providing formally precise blueprints of the constraint propagation mechanisms required, we may both promote an understanding and facilitate the adoption of our constraints in other systems, not necessarily based, as ours, on the concurrent constraint programming language Oz [19].

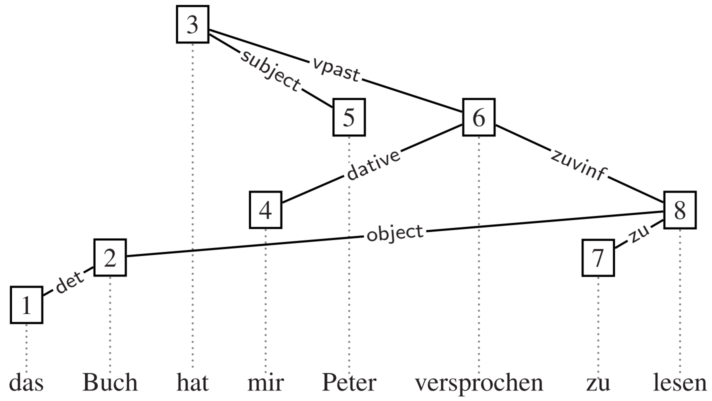
We begin the article, in Section 2, with an example of dependency analysis for German taken from [4]: it provides a concrete illustration that helps understand the abstractions which follow. Section 3 lays down the foundation in the form of a formalization of labeled trees. To a great extent, the conciseness of our formulation and the effectiveness of our treatment of ambiguity rely on 'selection constraints': first introduced in [4], they are presented intuitively in Section 4 and further extended to an entire family of aggregative selection constraints.

We then consider various refinements of interest to the computational linguist: in Section 5 we tackle the problem of lexical ambiguity; in Section 6 we further extend it to the treatment of lexicalized valency constraints; in Section 7 we describe how families of well-formedness principles can be expressed and integrated into our framework to further restrict admissibility of edges, e.g. for reasons of 'agreement'; in Section 8 we introduce 'disjunctive attributes' for a treatment of ambiguity that helps improve lexical economy; then in Section 9 we propose an extension for partially ordered trees which permits the characterization of projective analyses and the formulation of linear precedence constraints.

Finally in Section 11 we address the issue of efficient processing and describe what we expect from the constraint programming services. We do not commit to any particular implementation technique, rather we describe constraint programming at a very abstract level as a formal system with deterministic inference rules of 'propagation' and non-deterministic rules of 'distribution'. In this framework, the search for solutions is precisely the derivation of consistent inferential saturations. We believe that we offer the right level of abstraction to permit instantiation to a variety of constraint programming and constraint logic programming systems, and the right level of detail for the practical implementation of our constraints should they not already exist in the target system.

2 Informal Introduction to Dependency Parsing

In [4] we described a constraint-based approach for constructing dependency tree analyses of German sentences: it serves as the starting point for the more general and abstract formulation developed in the remainder of the present article. In this section, we begin with an example taken from that earlier work to provide the reader with a concrete illustration of what we propose to generalize.



$$\boxed{2}\begin{bmatrix} \mathsf{str} : \texttt{"Buch"} \\ \mathsf{cat} : \texttt{n} \\ \mathsf{agr} : \langle \texttt{neut sing 3 acc} \rangle \\ \mathsf{val} : \{\texttt{det}\} \end{bmatrix} \qquad \boxed{3}\begin{bmatrix} \mathsf{str} : \texttt{"hat"} \\ \mathsf{cat} : \texttt{v\_fin} \\ \mathsf{agr} : \langle \texttt{neut sing 3 nom} \rangle \\ \mathsf{val} : \{\texttt{subject}, \texttt{vpast}\} \end{bmatrix}$$

FIG. 1. Example of Dependency Tree

Consider the German sentence *"das Buch hat mir Peter versprochen zu lesen"*.[1] Figure 1 shows a corresponding dependency analysis and illustrates the fact that the relative freeness of word order often requires analyses with crossing branches. Characteristic of the dependency approach, there is exactly one node per word, displayed here as a box labeled with the linear position of the word in the sentence, and edges are labeled with syntactic roles such as '`subject`' or '`zuvinf`'. Furthermore each node is assigned a lexical entry from a lexicon: Figure 1 displays the lexical entries assigned by the parser to '`Buch`' and '`hat`' identified respectively by $\boxed{2}$ and $\boxed{3}$. Each entry stipulates phonology, category, agreement and valency. In Section 6, we propose an alternative approach to lexicalized valency constraints.

Dependency Grammar. We now briefly review the formal framework for dependency grammar proposed in [4]. For simplicity, we omit the treatment of modifiers[2] and of linear precedence.[3] A dependency grammar is given by:

$$(Strs, Cats, Agrs, Roles, Lexicon, Rules)$$

where *Strs* is a finite set of strings, such as `"Buch"` or `"hat"`, notating the fully inflected forms of words, *Cats* is a finite set of categories such as `n` for noun, `d` for

[1] Stripped of intonation and of modifiers, for the sake of simplicity, the example does not sound convincing to the German ear, but the following sentence due to Joachim Niehren exhibits the same structure and sounds perfectly natural: *"Genau diese Flasche Wein hat mir mein Kommissionnär versprochen auf der Auktion zu ersteigern"*.

[2] We shall see in Section 6 that modifiers do not actually require separate treatment in our approach.

[3] We consider it again in Section 9 and more extensively in [6].

determiner, or v_fin for finite verb. Assuming the following sets for gender, number, person and case:

$$\mathit{Gender} = \{\texttt{masc},\ \texttt{fem},\ \texttt{neut}\} \qquad \mathit{Number} = \{\texttt{sing},\ \texttt{plur}\}$$
$$\mathit{Person} = \{\texttt{1},\ \texttt{2},\ \texttt{3}\} \qquad \mathit{Case} = \{\texttt{nom},\ \texttt{acc},\ \texttt{dat},\ \texttt{gen}\}$$

we pose $\mathit{Agrs} = \mathit{Gender} \times \mathit{Number} \times \mathit{Person} \times \mathit{Case}$ for the set of agreement tuples such as $\langle \texttt{masc sing 3 nom} \rangle$. *Roles* is a finite set of grammatical functions such as `subject` or `zuvinf` for an infinitive with 'zu' which serve as edge labels in the dependency tree. A lexical entry is an attribute value matrix with signature:

$$\begin{bmatrix} \mathsf{str} & : & \mathit{Strs} \\ \mathsf{cat} & : & \mathit{Cats} \\ \mathsf{agr} & : & \mathit{Agrs} \\ \mathsf{val} & : & 2^{\mathit{Roles}} \end{bmatrix}$$

and specifies phonology, category, agreement and valency. The *Lexicon* is a finite set of lexical entries. We use functional notation and write $\mathsf{cat}(w)$ for the category of the lexical entry assigned to node w. Finally, *Rules* is a family (Γ_ρ) of binary predicates, indexed by grammatical functions $\rho \in \mathit{Roles}$, expressing local grammatical principles: for an edge labeled ρ from mother w_1 to daughter w_2 to be admissible, the condition $\Gamma_\rho(w_1, w_2)$ must be satisfied; for example, for w_2 to serve as the determiner of noun w_1 it must (1) be a determiner, (2) agree with the noun, i.e.:

$$\Gamma_{\texttt{det}}(w_1, w_2) \quad \equiv \quad \mathsf{cat}(w_2) = \mathsf{d} \ \wedge \ \mathsf{agr}(w_1) = \mathsf{agr}(w_2)$$

Dependency Trees. We assume an infinite set *Nodes* of nodes and define a labeled directed edge as an element of $\mathit{Nodes} \times \mathit{Nodes} \times \mathit{Roles}$. A dependency tree (V, E, entry) consists of a finite set $V \subseteq \mathit{Nodes}$ of nodes, a finite set $E \subseteq V \times V \times \mathit{Roles}$ of labeled edges between these nodes, and a function $\mathsf{entry} : V \to \mathit{Lexicon}$ assigning a lexical entry to each node. A dependency tree is admissible iff (1) it forms a tree in the classical graph theoretical sense, (2) every node has precisely the outgoing edges required by its valency, (3) for every edge $(w_1, w_2, \rho) \in E$, the condition $\Gamma_\rho(w_1, w_2)$ is satisfied.

Of course, the lexicon typically contains several lexical entries for each word which results in considerable lexical ambiguity. Our approach is very effective in handling both lexical and structural ambiguity and achieves this largely through the use of 'selection constraints' (Section 4). Figure 2 illustrates the processing achieved for our example by the parser of [4]. On the left is displayed the preferred reading, while on the right we see the complete search tree where a circle represents a choice point, a diamond leaf a solution, and a square leaf a failure. What is interesting is that there are no failures: constraint propagation is very effective; we need exactly 1 choice in order to enumerate the two possible analyses. Why are there 2 analyses? Simply because both 'Buch' and 'Peter' can indifferently be assigned either nominative or accusative case, therefore either one can be subject while the other is object. Constraint propagation is sufficient to resolve all other ambiguities, both lexical (what lexical entry to choose for each word), and structural (what edges to draw between nodes).

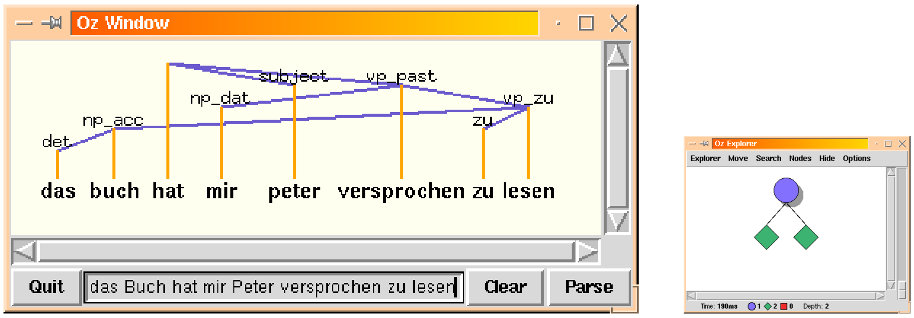


FIG. 2. Parser Demo

3 Formalization of Valid Labeled Trees

We begin with a precise characterization of all well-formed trees which can be assembled from a fixed finite set of nodes V and edges with labels drawn from a finite set $\mathcal{L}$. These trees correspond precisely to the solutions of the constraint satisfaction problem on set variables articulated below.

Finite Labeled Graphs. We assume given an infinite set $\mathcal{V}$ of nodes and a finite set $\mathcal{L}$ of labels. A directed labeled edge is an element of $\mathcal{V} \times \mathcal{V} \times \mathcal{L}$. We write $\mathbf{G}(\mathcal{V}, \mathcal{L})$ for the set of finite graphs $G = (V, E)$ formed from a finite set of nodes $V \subseteq \mathcal{V}$ and a finite set of labeled edges $E \subseteq V \times V \times \mathcal{L}$. Note that, since we assume E to be a set, we only consider graphs without duplicate edges. We write $\mathbf{G}(V, \mathcal{L})$ for the graphs in $\mathbf{G}(\mathcal{V}, \mathcal{L})$ whose node set is V.

Finite Labeled Trees. A finite graph is a tree if and only if it satisfies the following "treeness conditions":

(a) Each node has at most one incoming edge
(b) There is precisely one node with no incoming edge (one root)
(c) There are no cycles

We write $\mathbf{T}(V, \mathcal{L})$ for the subset of $\mathbf{G}(V, \mathcal{L})$ satisfying conditions (a), (b) and (c). In the following, we are going to formulate a constraint $\mathbf{\Phi}_{\mathrm{ID}}(V, \mathcal{L})$ which a finite graph $G = (V, E) \in \mathbf{G}(V, \mathcal{L})$ must satisfy in order to be in $\mathbf{T}(V, \mathcal{L})$.

We write $w{-}\ell{\rightarrow}w'$ for a labeled edge (w, w', ℓ) and $w{-}\ell{\rightarrow}_G w'$ for $w{-}\ell{\rightarrow}w' \in E$. We define the successor relation $\rightarrow_G = \cup\{-\ell{\rightarrow}_G \mid \ell \in \mathcal{L}\}$ and write $\rightarrow_G^+$ and $\rightarrow_G^*$ for its transitive and reflexive transitive closures. Given a relation $R \subseteq V \times V$, we define $R : V \to 2^V$ and the overloading $R : 2^V \to 2^V$ as follows:

$$R(x) = \{y \mid (x, y) \in R\} \qquad\qquad R(S) = \cup\{R(x) \mid x \in S\}$$

In this manner, the edges of a graph G induce the following functions:

$$\begin{array}{rlrlrl} \ell_G = & -\ell\!\to_G & \mathsf{down}_G = & \to^+_G & \mathsf{roots}_G = & V \setminus \to_G(V) \\ \mathsf{daughters}_G = & \to_G & \mathsf{eqdown}_G = & \to^*_G & & \end{array}$$

Given these definitions, we can reformulate the treeness conditions more formally. Condition (a) states that each node has at most one incoming edge, i.e. that for any node w'', there exists at most one $\ell \in \mathcal{L}$ and one $w \in V$, such that $w'' \in \ell_G(w)$; or, equivalently, that:

$$\forall \ell, \ell' \in \mathcal{L}\ \ \forall w, w' \in V \qquad (\ell \neq \ell'\ \vee\ w \neq w')\ \Rightarrow\ \ell_G(w) \parallel \ell'_G(w') \tag{3.1}$$

where $\parallel$ represents disjointness. Condition (b) requires that there be one unique root:

$$|\mathsf{roots}_G| = 1 \tag{3.2}$$

Finally, condition (c) forbids cycles, i.e. it must never be the case that $w \to^+_G w$:

$$\forall w \in V \qquad w \notin \mathsf{down}_G(w) \tag{3.3}$$

Our formalization so far assumes that the edges are given. As such, it is appropriate for deciding whether a graph is a tree. On the other hand, it is poorly suited for parsing where the edges are unknown and the task is precisely to find possible edges licensed by the grammar. To overcome this difficulty, instead of using the edges as a starting point, we are going to use the functions defined above which they induce.

In a tree, these functions satisfy additional properties which we state below. $\ell_G(w)$ are the ℓ-daughters of w. By definition of $\to_G$ and condition (a) restated as (3.1), the daughters of w satisfy the equation:

$$\mathsf{daughters}_G(w) = \uplus\{\ell_G(w) \mid \ell \in \mathcal{L}\} \tag{3.4}$$

where $\uplus$ denotes disjoint union. By definition of roots_G and condition (a), a node is either a root or is the daughter of precisely one node:

$$V = \mathsf{roots}_G\ \uplus\ \uplus\{\mathsf{daughters}_G(w) \mid w \in V\} \tag{3.5}$$

By definition of transitive closure: $\to^+_G\ =\ \to_G \circ \to^*_G$. In other words, the nodes strictly below w are those equal to or strictly below its daughters:

$$\mathsf{down}_G(w) = \cup\{\mathsf{eqdown}_G(w') \mid w' \in \mathsf{daughters}_G(w)\} \tag{3.6}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{\Phi}_{\mathsf{ID}}(V,\mathcal{L}) \ \equiv} \\
 & \multicolumn{2}{l}{V = \mathsf{roots} \ \uplus \ \uplus\{\mathsf{daughters}(w) \mid w \in V\}} \\
\wedge & \multicolumn{2}{l}{|\mathsf{roots}| = 1} \\
\wedge & \multicolumn{2}{l}{\forall w \in V} \\
 & & \mathsf{eqdown}(w) = \{w\} \uplus \mathsf{down}(w) \\
 & \wedge & \mathsf{down}(w) = \cup\{\mathsf{eqdown}(w') \mid w' \in \mathsf{daughters}(w)\} \\
 & \wedge & \mathsf{daughters}(w) = \uplus\{\ell(w) \mid \ell \in \mathcal{L}\}
\end{array}$$

FIG. 3. well-formedness condition of labeled trees

By definition of reflexive transitive closure: $\rightarrow^*_G(w) = \{w\} \cup \rightarrow^+_G(w)$. Additionally the acyclicity condition (c) requires that w does not occur in $\rightarrow^+_G(w)$. Therefore:

$$\mathsf{eqdown}_G(w) = \{w\} \uplus \mathsf{down}_G(w) \tag{3.7}$$

These properties lead us to formulate a constraint $\mathbf{\Phi}_{\mathsf{ID}}(V,\mathcal{L})$ (see Figure 3) in terms of variable roots of type 2^V and functional variables daughters, down, eqdown and ℓ (for all $\ell \in \mathcal{L}$) of type $V \rightarrow 2^V$. A solution of $\mathbf{\Phi}_{\mathsf{ID}}(V,\mathcal{L})$ is an assignment to these variables such that $\mathbf{\Phi}_{\mathsf{ID}}(V,\mathcal{L})$ is satisfied; in other words, posing $\mathcal{L} = \{\ell_1,\ldots,\ell_n\}$ and writing F for the type $V \rightarrow 2^V$, a solution of $\mathbf{\Phi}_{\mathsf{ID}}(V,\mathcal{L})$ is a tuple:

$$(\mathsf{roots}, \mathsf{daughters}, \mathsf{down}, \mathsf{eqdown}, \ell_1,\ldots,\ell_n) \ : \ 2^V \times F \times F \times F \times F \times \cdots \times F$$

that satisfies $\mathbf{\Phi}_{\mathsf{ID}}(V,\mathcal{L})$. We write $\mathit{Sols}(\mathbf{\Phi}_{\mathsf{ID}}(V,\mathcal{L}))$ for the set of solutions of $\mathbf{\Phi}_{\mathsf{ID}}(V,\mathcal{L})$. Every solution σ of $\mathbf{\Phi}_{\mathsf{ID}}(V,\mathcal{L})$ defines a graph $[\![\sigma]\!]_{\mathsf{ID}} = (V,E)$ where:

$$E = \{w{-}\ell{\rightarrow}w' \mid w \in V,\ \ell \in \mathcal{L},\ w' \in \ell(w)\}$$

Overloading the notation, for each $G \in \mathbf{G}(V,\mathcal{L})$ we also define:

$$[\![G]\!]_{\mathsf{ID}} = (\mathsf{roots}_G, \mathsf{daughters}_G, \mathsf{down}_G, \mathsf{eqdown}_G, \ell_{1G},\ldots,\ell_{nG})$$

THEOREM 3.1
$\mathbf{T}(V,\mathcal{L})$ *is in bijection with the solutions of* $\mathbf{\Phi}_{\mathsf{ID}}(V,\mathcal{L})$*. More precisely a graph G is a tree iff* $[\![G]\!]_{\mathsf{ID}}$ *satisfies* $\mathbf{\Phi}_{\mathsf{ID}}(V,\mathcal{L})$*:*

$$\forall G \in \mathbf{G}(V,\mathcal{L}) \quad G \in \mathbf{T}(V,\mathcal{L}) \ \equiv \ [\![G]\!]_{\mathsf{ID}} \in \mathit{Sols}(\mathbf{\Phi}_{\mathsf{ID}}(V,\mathcal{L}))$$

and every solution σ *of* $\mathbf{\Phi}_{\mathsf{ID}}(V,\mathcal{L})$ *defines a tree:*

$$\forall \sigma \in \mathit{Sols}(\mathbf{\Phi}_{\mathsf{ID}}(V,\mathcal{L})) \quad [\![\sigma]\!]_{\mathsf{ID}} \in \mathbf{T}(V,\mathcal{L})$$

The first claim follows from properties (3.4–3.7), while for the second one it is straightforward to establish $\sigma = [\![[\![\sigma]\!]_{\mathsf{ID}}]\!]_{\mathsf{ID}}$. $\mathbf{\Phi}_{\mathsf{ID}}(V,\mathcal{L})$ can be interpreted as a constraint satisfaction problem (CSP) and Theorem 3.1 establishes the correspondence between the solutions of this CSP and the formal objects of interest, namely the trees $\mathbf{T}(V,\mathcal{L})$. Throughout this article, we state similar theorems to validate the constraint-based approach.

4 Set Constraints And Selection Constraints

Equation (3.6) while mathematically elegant poses a processing challenge: when solving the CSP, (1) we don't know the elements of $\mathsf{daughters}(w)$, (2) we don't know what set values eqdown takes at these elements, and yet we must compute their combined union. In this section, we introduce the constraint-based concepts that will allow us to achieve it simply and efficiently.

4.1 Set Constraints

Finite domain (FD) constraints have become a reasonably standard tool of the trade and are routinely used in computational linguistics applications. Set constraints, on the other hand, have remained largely unexploited even though they are available and well supported by modern constraint technology [12, 20, 18].

In our work, constraints on finite sets (FS) of integers have emerged as an especially elegant and computationally effective tool for such linguistics applications as parsing with a dependency grammar [4, 6] or solving dominance constraints [5, 8], for the treatment of discourse [7], parsing with tree descriptions [9], and underspecified representations of semantics [10].

We will elaborate on constraint programming at greater length in Section 11. For the moment, we shall simply say that the partial information about a FD variable I can be represented in the form $I \in D$ where D is a set of integers, and the partial information about a FS variable S can be expressed by a lower bound D_1 and an upper bound D_2 in the form $D_1 \subseteq S \subseteq D_2$. The role of constraint propagation is to improve this partial information. When $I \in \{k\}$ we say that I is 'determined' and write $I = k$. When $D \subseteq S \subseteq D$, we say that S is 'determined' and write $S = D$.

4.2 Selection Constraints

An essential contribution of [4] was the 'selection constraint' which permits the compact representation and effective processing of many forms of selectional ambiguity such as lexical ambiguity (i.e. the selection of a lexical entry from those available for a particular word in the lexicon). Consider a variable X which may be equated with one of n variables (V_i). We can explicitly represent this choice using an integer variable I, also called a finite domain (FD) variable, taking values in $\{1 \ldots n\}$ and the selection constraint below:

$$X = \langle V_1, \ldots, V_n \rangle[I]$$

where $\langle V_1, \ldots, V_n \rangle$ represents the sequence of variables V_1 through V_n and the notation $\langle V_1, \ldots, V_n \rangle[I]$ was chosen for its similarity to the subscripting notation of 'array lookup' in many programming languages, and indicates selection of the Ith element out of the sequence. Thus the declarative semantics of the above constraint is simply $X = V_I$.

The origins of this powerful idea are to be found in CHIP's 'element' constraint [2] which related two finite domain variables I and K and a sequence $\langle j_1, \ldots, j_n \rangle$ of integer values:

$$I = \langle j_1, \ldots, j_n \rangle [K]$$

In [4], we extended it in two directions: first we allowed the sequence to consist of variables rather than constants; second, we supported both selection out of sequences of finite domain variables:

$$I = \langle J_1, \ldots, J_n \rangle [K]$$

as well as out of sequences of finite set (FS) variables:

$$S = \langle S_1, \ldots, S_n \rangle [K]$$

where S, S_i are FS variables denoting finite sets of integers.

4.3 Propagation And Constructive Disjunction

One advantage of the selection constraint is that it is able to implement simply and efficiently a form of constructive disjunction (lifting of information common to all alternatives not yet ruled out). Here is an example that illustrates the propagation which may be expected from the selection constraint:

$$S = \langle S_1, S_2, S_3 \rangle [K] \quad \{1,3\} \subseteq S_1 \subseteq \{1,2,3\} \quad S_2 = \{2,4\} \quad \{1\} \subseteq S_3 \subseteq \{1,4\}$$

From the above, constraint propagation infers:

$$K \in \{1,2,3\} \qquad S \subseteq \{1,2,3,4\}$$

If we further assert $4 \notin S$, then S_2 becomes incompatible since it contains 4:

$$K \in \{1,3\} \qquad \{1\} \subseteq S \subseteq \{1,2,3\}$$

Note that 1 was inferred to be a necessary element of S since it is a known element of both alternatives S_1 and S_3, one of which must eventually be chosen. If we now assert $2 \in S$, then S_3 becomes incompatible since it cannot contain 2:

$$K = 1 \qquad S = S_1 = \{1,2,3\}$$

4.4 Dependent Disjunction

The fact that the selection constraints makes the choice explicit through a selector variable K permits dependent selections. Consider for example:

$$I = \langle J_1, \ldots, J_n \rangle [K] \tag{4.1}$$

$$I' = \langle J'_1, \ldots, J'_n \rangle [K] \tag{4.2}$$

the choice of which J_i to equate with I and which J'_i to equate with I' are mutually dependent since they must be effected by the same selector K. (4.1) and (4.2) can be viewed as contexted constraints sharing the same context variable K, or equivalently as realizing the following dependent (or named) disjunctions both labeled with name I [17, 3, 11, 13]:

$$\begin{array}{l} (I\ = J_1 \vee \quad \dots \quad \vee\ I\ = J_n)_I \\ (I' = J'_1 \vee \quad \dots \quad \vee\ I' = J'_n)_I \end{array}$$

Notational variations on dependent disjunctions have been used to concisely express covariant assignment of values to different features in feature structures. The selection constraint provides the same notational convenience and declarative semantics, but additionally enjoys a computational reading with all the benefits that accrue from state-of-the-art constraint technology.

4.5 Selection Union Constraint

A novel contribution of the present article is the ‘selection union’ constraint:

$$S = \cup\langle S_1, \dots, S_n\rangle[S']$$

where the selector S' is now a set. Its declarative semantics are given by the following equation:

$$S = \cup\{S_k \mid k \in S'\}$$

Thus we empower an essential mathematical instrument with a computational reading based on very effective constraint propagation. Posing $V = \{w_1, \dots, w_n\}$ and identifying w_i with the integer i representing its position in the input sentence,[4] equation (3.6) can now be rewritten:

$$\mathsf{down}(w) = \cup\langle \mathsf{eqdown}(w_1), \dots, \mathsf{eqdown}(w_n)\rangle[\mathsf{daughters}(w)]$$

A family of aggregative selection constraints. The selection union constraint selects a subset specified by S' of the elements of sequence $\langle S_1, \dots, S_n\rangle$ and combines them using union. Clearly, other modes of combination are possible which opens up a whole family of aggregative selection constraints. For example:

$$\begin{array}{c} S = \cap\langle S_1, \dots, S_n\rangle[S'] \\ S = \uplus\langle S_1, \dots, S_n\rangle[S'] \\ I = +\langle J_1, \dots, J_n\rangle[S'] \end{array}$$

In Section 11 we take the ‘selection union’ and ‘selection intersection’ constraints as primitives and fully specify their operational semantics by inference rules of propagation. The simpler selection constraints of Section 4.2 are defined in terms of the selection union constraint.

[4] This identification of a node with the linear position of the corresponding word in the input sentence will remain in effect for the remainder of the article.

5 Lexical Ambiguity

For the application to dependency parsing, each node is assigned a lexical entry from a lexicon. A lexical entry supplies a number of attributes in terms of which additional constraints may be formulated. In this section, we formalize these notions and demonstrate how the selection constraint elegantly addresses the issue of lexical ambiguity.

A lexicon $(\mathcal{E}, \mathcal{A})$ consists of a finite set $\mathcal{E}$ of objects called 'lexical entries' and a finite set $\mathcal{A}$ of functions called 'attributes', where each $\alpha \in \mathcal{A}$ is of type $\alpha : \mathcal{E} \to \mathbb{N}$ or $\alpha : \mathcal{E} \to 2^{\mathbb{N}}$. For each $e \in \mathcal{E}$ and $\alpha \in \mathcal{A}$, $\alpha(e)$ is the value of attribute α in lexical entry e. An attribute might specify such things as category, agreement, or valency.

Given a lexicon $(\mathcal{E}, \mathcal{A})$, a graph $(V, E) \in \mathbf{G}(V, \mathcal{L})$, and an assignment $\varepsilon : V \to \mathcal{E}$ of lexical entries to nodes, we call (V, E, ε) an *attributed graph*. We write $\mathbf{G}(V, \mathcal{L}, \mathcal{E}, \mathcal{A})$ for the set of attributed graphs and $\mathbf{T}(V, \mathcal{L}, \mathcal{E}, \mathcal{A})$ for the subset which are trees. For each $\alpha \in \mathcal{A}$ with type $\alpha : \mathcal{E} \to T$, where T is $\mathbb{N}$ or $2^{\mathbb{N}}$, we introduce the overloaded function $\alpha : V \to T$ called a *node attribute* and defined by $\alpha(w) = \alpha(\varepsilon(w))$ for all $w \in V$.

For parsing, the choice of ε is not free: in particular, only a subset of $\mathcal{E}$ is applicable to each word. This we model by means of a restriction function $\mathsf{lex} : V \to 2^{\mathcal{E}}$. For example, if w corresponds to the word 'versprochen', then $\mathsf{lex}(w)$ should be the set of lexical entries for 'versprochen'. We say that (V, E, ε) is lex-attributed if:

$$\forall w \in V \;\; \varepsilon(w) \in \mathsf{lex}(w) \tag{5.1}$$

and write $\mathbf{T}(V, \mathcal{L}, \mathcal{E}, \mathcal{A}, \mathsf{lex})$ for the set of lex attributed trees (V, E, ε) over $(\mathcal{E}, \mathcal{A})$. During parsing, ε is not given but must be chosen. The degree of freedom in this choice (see 5.1) is called lexical ambiguity. In order to support efficient parsing, it should be possible to constrain a node attribute $\alpha(w)$ while leaving the choice $\varepsilon(w)$ of lexical entry underspecified.

Thus we are confronted with the problem of computing a function at a point which is only partially known. Supposing $\mathsf{lex}(w) = \{e_1, \ldots, e_n\}$, the idea is to introduce an FD variable $\mathsf{entry}(w) \in \{1, \ldots, n\}$ to represent the index of the selected entry and to obtain $\alpha(w)$ with the following selection constraint:

$$\alpha(w) = \langle \alpha(e_1), \ldots, \alpha(e_n) \rangle [\mathsf{entry}(w)] \tag{5.2}$$

In this equation, the sequence consists of elements homogeneously of type $\mathbb{N}$ or $2^{\mathbb{N}}$ and thus is in the domain of applicability of the selection constraint. For $w \in V$, we have one equation (5.2) for each $\alpha \in \mathcal{A}$, but they all share the same selector $\mathsf{entry}(w)$. In this fashion, as explained in Section 4.4, all attribute selections for the same node are forced to be covariant. This is an additional source of effective propagation: if any constraint affects one selection, it affects them all.

It is this intuition which we now proceed to formalize. Without loss of generality, we revise the type of $\mathsf{lex} : V \to \mathcal{E}^*$ to map each node to a sequence rather than a set of lexical entries, and assume that each sequence contains no duplicates. We define:

$$e = \langle e_1, \ldots, e_n \rangle[k] \quad \equiv \quad 1 \leq k \leq n \ \wedge \ e = e_k$$

We write $e \in \mathsf{lex}(w)$ when e occurs in sequence $\mathsf{lex}(w)$ and denote by $e @ \mathsf{lex}(w)$ its position in this sequence, or more generally the smallest i such that $e = \mathsf{lex}(w)[i]$. An assignment $\varepsilon : V \to \mathcal{E}$ is lex-restricted iff:

$$\forall w \in V \quad \varepsilon(w) \in \mathsf{lex}(w) \tag{5.3}$$

ε induces functions $\mathsf{entry}_\varepsilon : V \to \mathbb{N}$ and $\alpha_{i\varepsilon} : V \to T_i$ for each $\alpha_i : \mathcal{E} \to T_i \ \in \ \mathcal{A}$:

$$\begin{aligned} \mathsf{entry}_\varepsilon(w) &= \varepsilon(w) @ \mathsf{lex}(w) \\ \alpha_{i\varepsilon}(w) &= \alpha_i(\varepsilon(w)) \end{aligned}$$

Given a sequence $\langle e_1 \ldots e_n \rangle \in \mathcal{E}^*$, we define:

$$\langle \alpha \mid \langle e_1 \ldots e_n \rangle \rangle \quad = \quad \langle \alpha(e_1) \ldots \alpha(e_n) \rangle$$

The functions induced by ε satisfy the following property:

$$\forall w \in V, \ \forall \alpha_i \in \mathcal{A} \qquad \alpha_{i\varepsilon} \ = \ \langle \alpha_i \mid \mathsf{lex}(w) \rangle [\mathsf{entry}_\varepsilon(w)] \tag{5.4}$$

Posing $\mathcal{A} = \{\alpha_1 : \mathcal{E} \to T_1, \ldots, \alpha_n : \mathcal{E} \to T_n\}$ (where T_i is $\mathbb{N}$ or $2^{\mathbb{N}}$), we now formulate a constraint $\mathbf{\Phi}_{\mathsf{LEX}}(V, \mathcal{E}, \mathcal{A}, \mathsf{lex})$ (Figure 4) in terms of functional variables $\mathsf{entry} : V \to \mathbb{N}$ and $\alpha_i : V \to T_i$ for $\alpha_i : \mathcal{E} \to T_i \in \mathcal{A}$. $\mathbf{\Phi}_{\mathsf{LEX}}(V, \mathcal{E}, \mathcal{A}, \mathsf{lex})$ characterizes all lex-restricted assignments $\varepsilon : V \to \mathcal{E}$ over $(\mathcal{E}, \mathcal{A})$.

$$\begin{aligned} &\mathbf{\Phi}_{\mathsf{LEX}}(V, \mathcal{E}, \mathcal{A}, \mathsf{lex}) \ \equiv \\ &\quad \forall w \in V \quad \bigwedge_{\alpha \in \mathcal{A}} \alpha(w) = \langle \alpha \mid \mathsf{lex}(w) \rangle [\mathsf{entry}(w)] \end{aligned}$$

FIG. 4. Well-formedness condition for lex-restricted assignments

A solution of $\mathbf{\Phi}_{\mathsf{LEX}}(V, \mathcal{E}, \mathcal{A}, \mathsf{lex})$ is a tuple:

$$(\mathsf{entry}, \alpha_1, \ldots, \alpha_n) : (V \to \mathbb{N}) \times (V \to T_1) \times \cdots \times (V \to T_n)$$

that satisfies $\mathbf{\Phi}_{\mathsf{LEX}}(V, \mathcal{E}, \mathcal{A}, \mathsf{lex})$. Every solution σ of $\mathbf{\Phi}_{\mathsf{LEX}}(V, \mathcal{E}, \mathcal{A}, \mathsf{lex})$ defines a lex-restricted assignment $[\![\sigma]\!]_{\mathsf{LEX}}$:

$$[\![\sigma]\!]_{\mathsf{LEX}}(w) = \mathsf{lex}(w)[\mathsf{entry}(w)]$$

Furthermore, for each lex-restricted assignment ε, we also define:

$$[\![\varepsilon]\!]_{\mathsf{LEX}} = (\mathsf{entry}_\varepsilon, \alpha_{1\varepsilon}, \ldots, \alpha_{n\varepsilon})$$

Writing $\mathbf{A}(V, \mathcal{E}, \mathcal{A}, \mathsf{lex})$ for the set of lex-restricted assignments $\varepsilon : V \to \mathcal{E}$ over $(\mathcal{E}, \mathcal{A})$, we have:

THEOREM 5.1
$\mathbf{A}(V, \mathcal{E}, \mathcal{A}, \mathsf{lex})$ *is in bijection with Sols*$(\mathbf{\Phi}_{\mathsf{LEX}}(V, \mathcal{E}, \mathcal{A}, \mathsf{lex}))$.

$$\begin{array}{ll} \forall \sigma \in \mathit{Sols}(\mathbf{\Phi}_{\mathsf{LEX}}(V, \mathcal{E}, \mathcal{A}, \mathsf{lex})) & [\![\sigma]\!]_{\mathsf{LEX}} \in \mathbf{A}(V, \mathcal{E}, \mathcal{A}, \mathsf{lex}) \\ \forall \varepsilon \in \mathbf{A}(V, \mathcal{E}, \mathcal{A}, \mathsf{lex}) & [\![\varepsilon]\!]_{\mathsf{LEX}} \in \mathit{Sols}(\mathbf{\Phi}_{\mathsf{LEX}}(V, \mathcal{E}, \mathcal{A}, \mathsf{lex})) \end{array}$$

6 Structural Ambiguity and Valency Constraints

In the constraint formulation $\mathbf{\Phi}_{\mathsf{ID}}(V, \mathcal{L})$ of Figure 3, for each $w \in V$ and $\ell \in \mathcal{L}$, $\ell(w)$ is a set variable. During parsing, this variable is typically partially known. For example it might be constrained by the following bounds:

$$\{w_1, w_2\} \subseteq \ell(w) \subseteq \{w_1, w_2, w_3\}$$

indicating that w's only possible outgoing edges labeled with ℓ are $w{-}\ell{\rightarrow}w_1$, $w{-}\ell{\rightarrow}w_2$ and $w{-}\ell{\rightarrow}w_3$, that the first two have been accepted, but that the case of the last one hasn't been decided yet. Thus, with set variables, we are able to represent an ambiguous tree structure. All possible edges are simultaneously represented, initially $\emptyset \subseteq \ell(w) \subseteq V$, and parsing is a process of disambiguation: candidate edges are either accepted or rejected. When every $\ell(w)$ is determined, the tree is fully disambiguated.

Disambiguation is to a large extent driven by grammatical constraints for subcategorization embodied in lexicalized valency constraints.

Lexicalized Valency Constraints. When parsing with a dependency grammar [4], we are not free to draw arbitrary edges between nodes. The outgoing edges of a node represent the complements and modifiers of the corresponding word. The nature and number of these edges are restricted by grammatical valencies specified in the lexicon. We formalize this as follows:

For every $\ell \in \mathcal{L}$, there is a corresponding attribute $|\cdot|_\ell \in \mathcal{A}$ with type $|\cdot|_\ell : \mathcal{E} \rightarrow 2^{\mathbb{N}}$. Thus each lexical entry e stipulates a set $|e|_\ell$ of licensed cardinalities for the ℓ-daughter set: the number of outgoing edges of w labeled with ℓ must be one in $|w|_\ell$:

$$|\ell(w)| \in |w|_\ell \tag{6.1}$$

$|w|_\ell$ depends on the choice of lexical entry and, following equation (5.2), is given by the selection constraint:

$$|w|_\ell = \langle |\cdot|_\ell \mid \mathsf{lex}(w)\rangle[\mathsf{entry}(w)] \tag{6.2}$$

In order to specify a required ℓ-argument, a lexical entry e need only fix $|e|_\ell = \{1\}$. For an optional argument: $|e|_\ell = \{0, 1\}$. For a modifier which may appear 0 or any number of times: $|e|_\ell = \{0, 1, \ldots, \mu\}$ where μ is some arbitrarily large integer—for any particular sentence of length n, it is sufficient to choose $\mu = n - 1$ since for any one word, there are at most $n - 1$ arguments to be had. For an illegal argument: $|e|_\ell = \{0\}$.

We say that an attributed graph $G = (V, E, \varepsilon)$ fulfills its valencies iff:

$$\forall |\cdot|_\ell \in \mathcal{A},\ \forall w \in V \quad |\ell_G(w)| \in |\varepsilon(w)|_\ell \tag{6.3}$$

and formulate a corresponding constraint $\mathbf{\Phi}_{\mathsf{VAL}}(V, \mathcal{L}, \mathcal{E}, \mathcal{A})$ in Figure 5.

$$\mathbf{\Phi}_{\mathsf{VAL}}(V, \mathcal{L}, \mathcal{E}, \mathcal{A}) \quad \equiv \quad \forall w \in V \quad \bigwedge_{\ell \in \mathcal{L}} |\ell(w)| \in |w|_\ell$$

FIG. 5. Well-formedness constraint of valency-fulfilling graphs

Writing $\mathbf{T}_{\mathsf{VAL}}(V, \mathcal{L}, \mathcal{E}, \mathcal{A}, \mathsf{lex})$ for the set of lex-attributed trees that fulfill their valencies, we have:

THEOREM 6.1
$\mathbf{T}_{\mathsf{VAL}}(V, \mathcal{L}, \mathcal{E}, \mathcal{A}, \mathsf{lex})$ *is in bijection with the solutions of:*

$$\mathbf{\Phi}_{\mathsf{ID}}(V, \mathcal{L}) \wedge \mathbf{\Phi}_{\mathsf{LEX}}(V, \mathcal{E}, \mathcal{A}, \mathsf{lex}) \wedge \mathbf{\Phi}_{\mathsf{VAL}}(V, \mathcal{L}, \mathcal{E}, \mathcal{A})$$

Partial Functions. In passing, it may be useful to mention a related and generally useful transformation for the constraint-based treatment of partial functions. A partial function $f : A \to B$ is especially vexing for a constraint-based approach because a constraint about f is not meaningful everywhere, but only at points where f is defined. Frequently, the problem can be solved by replacing the partial function f by the total function $f' : A \to 2^B$ such that $f'(x) = \{f(x)\}$ if f is defined at x, and $f'(x) = \emptyset$ otherwise. For example, we might in this fashion conveniently model the notion of 'mother': every node has a unique mother except the root which has none.

7 Grammatical Principles Licensing Edges

While valency constraints restrict the number of edges for each grammatical function $\ell \in \mathcal{L}$, grammatical principles express additional local conditions for the admissibility of edges. For example, an `object` complement is required to be an accusative NP. In [4] (see Section 2) we proposed that a dependency grammar stipulate a family (Γ_ℓ) of binary predicates indexed by edge labels and such that $\Gamma_\ell(w, w')$ characterizes the grammatical admissibility of an edge $w{-}\ell{\rightarrow}w'$. In this section, we develop the framework for expressing these predicates and formalize the corresponding restrictions.

For our purposes, it suffices to consider the language whose abstract syntax is given in Figure 6 where x, y are variables ranging over nodes, i denotes an arbitrary integer, D an arbitrary finite set of integers, $\alpha(x)$ an attribute of node x, and $E \parallel E'$ expresses the disjointness of the sets denoted by E and E'.

For each $\ell \in \mathcal{L}$ there is a binary predicate Γ_ℓ of the form $\lambda x, y \cdot C$ (i.e. a P) which must be a closed abstraction of our language. We say that an attributed graph

$$
\begin{array}{rcll}
E & ::= & i \mid D \mid \alpha(x) & \\
C & ::= & \quad C \wedge C' & \\
 & & \mid \quad E < E' & \mid \quad E \leq E' \\
 & & \mid \quad E = E' & \mid \quad E \neq E' \\
 & & \mid \quad E \in E' & \mid \quad E \notin E' \\
 & & \mid \quad E \subseteq E' & \\
 & & \mid \quad E \parallel E' & \\
P & ::= & \lambda x, y \cdot C &
\end{array}
$$

FIG. 6. Constraint Language For Principles

$G = (V, E, \varepsilon)$ satisfies the grammatical principles (Γ_ℓ) if for all $w, w' \in V$ and $\ell \in \mathcal{L}$:

$$w' \in \ell(w) \;\Rightarrow\; G \models \Gamma_\ell(w, w') \tag{7.1}$$

where $G \models \Gamma_\ell(w, w')$ means that G satisfies $\Gamma_\ell(w, w')$ and is defined in the usual Tarskian way. For example, the grammatical principles validating an object complement or an adjective edge might be expressed as follows:

$$
\begin{array}{rcl}
\Gamma_{\texttt{object}} & \equiv & \lambda x, y \;\cdot\; \mathsf{cat}(y) \in \textsc{np} \;\wedge\; \mathsf{agr}(y) \in \textsc{acc} \\
\Gamma_{\texttt{adj}} & \equiv & \lambda x, y \;\cdot\; \mathsf{cat}(y) = \texttt{a} \;\wedge\; \mathsf{agr}(x) = \mathsf{agr}(y)
\end{array}
$$

where NP represents the set of noun phrase categories (e.g. $\{\texttt{n}, \texttt{pro}\}$), ACC the set of all agreements with accusative case, and a the category of adjectives. (7.1) gives rise to a quadratic number of implicational constraints which are expected to work in both direction, i.e. when $\Gamma_\ell(w, w')$ becomes inconsistent, $w' \notin \ell(w)$ should be inferred.

We formulate in Figure 7 the constraint $\mathbf{\Phi}_{\mathsf{P}}(V, \mathcal{L}, \mathcal{E}, \mathcal{A}, (\Gamma_\ell))$ which characterizes the attributed graphs that satisfy the grammatical principles (Γ_ℓ).

$$\mathbf{\Phi}_{\mathsf{P}}(V, \mathcal{L}, \mathcal{E}, \mathcal{A}, (\Gamma_\ell)) \quad \equiv \quad \forall w, w' \in V,\ \forall \ell \in \mathcal{L} \quad w' \in \ell(w) \Rightarrow \Gamma_\ell(w, w')$$

FIG. 7. well-formedness constraint for the satisfaction of grammatical principles

8 Disjunctive Attributes

Often it is convenient to use a set valued attribute $\alpha(w)$ to indicate a disjunction: any value in $\alpha(w)$ is licensed. We call α a disjunctive attribute. For example, distinct agreement values are frequently not morphologically distinguishable: while we could nonetheless create many otherwise identical lexical entries that differed only in agreement, in the interest of lexical economy it is convenient to use an attribute $\mathsf{agrs}(e)$ to represent the set of possible agreement values for lexical entry e.

For example, the German article “`den`” is either masculine singular accusative or any gender plural dative. However, only one value from this set may be assigned to the corresponding node. We formalize this notion as follows:

Let $\mathcal{A}_d \subseteq \mathcal{A}$ be a distinguished subset of so called ‘disjunctive’ attributes. They must be set-valued, i.e. for each $\alpha \in \mathcal{A}_d$, the lexical attribute has type $\alpha : \mathcal{E} \to 2^{\mathbb{N}}$. For each $\alpha \in \mathcal{A}_d$, there must exist a corresponding node attribute $\mathsf{the}_\alpha : V \to \mathbb{N}$ defined as follows:

$$\mathsf{the}_\alpha(w) \in \alpha(w) \tag{8.1}$$

For example, writing agr instead of $\mathsf{the}_{\mathsf{agrs}}$, we have:

$$\mathsf{agr}(w) \in \mathsf{agrs}(w)$$

which ensures that node w is assigned precisely one agreement value from the set of agreements licensed by its selected lexical entry. In practice, it is frequently legal for a parser to leave an attribute the_α underspecified. It is sound to so when it can be guaranteed that every partial solution that leaves the_α underspecified can be consistently extended to a complete solution that determines the_α. The conditions under which propagation is complete in this sense are outside the scope of this article.

9 Partially Ordered Projective Labeled Trees

The labeled trees considered so far are unordered and therefore cannot naturally express word order. In this section, we consider an extension to our formalization of labeled trees which overcomes this limitation.

Unordered labeled trees are adequate for the representation of ‘syntactic’ dependencies and sufficed for the treatment of immediate dependence presented in [4]. For a corresponding treatment of linear precedence, we proposed in [6] to introduce a second tree to represent ‘topological’ dependencies.

In the topological tree, both edges and nodes are labeled and the set of labels is totally ordered. As we shall see, thanks to this total order, it is possible to define the linearizations licensed by the tree. For concreteness, we present now an example:

dass	Maria	einen	Mann	wird	lieben	können[5]
that	*Maria*	*a*	*man*	*will*	*love*	*can*

Figure 8 displays its unordered non-projective syntax tree, while Figure 9 presents one possible topological tree. In the topological tree, edge labels are intended to correspond to the notion of *fields* in the classical topological sentence model [1], e.g.

[5] that Maria will be able to love a man

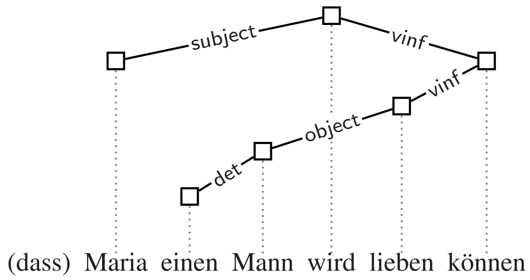


FIG. 8. Syntax tree

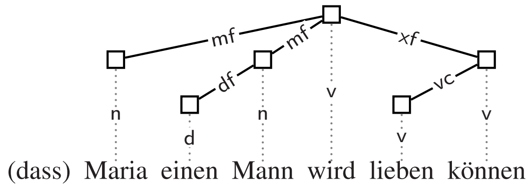


FIG. 9. Topological tree

mf represents the Mittelfeld and xf the extraposition field. Nodes are also labeled: in Figure 9 these labels are displayed on the vertical dotted lines joining a node to the word it stands for in the sentence. The total order assumed in the example is:

$$\mathsf{d} \prec \mathsf{df} \prec \mathsf{n} \prec \mathsf{mf} \prec \mathsf{vc} \prec \mathsf{v} \prec \mathsf{xf}$$

Order on edge labels induces a partial order on the daughters of each node and, by extension, on the subtrees rooted at these daughters. For example 'Mann' and 'können' have respectively edge labels mf and xf. Since $\mathsf{mf} \prec \mathsf{xf}$, 'einen Mann' precedes 'lieben können'. The order is partial because siblings with the same edge label are not respectively ordered: this is the basis for the account of scrambling in the Mittelfeld. The label assigned to a node allows to position it with respect to its daughters. For example 'wird' has node label v and, since $\mathsf{mf} \prec \mathsf{v} \prec \mathsf{xf}$, it must occur between 'einen Mann' and 'lieben können'. Thus the topological tree licenses 2 linearizations:

1. (dass) Maria einen Mann wird lieben können
2. (dass) einen Mann Maria wird lieben können

We will now distinguish a set $\mathcal{L}_\mathsf{E}$ of edge labels and a set $\mathcal{L}_\mathsf{N}$ of node labels, and assume given a total order $\prec$ on $\mathcal{L} = \mathcal{L}_\mathsf{E} \uplus \mathcal{L}_\mathsf{N}$. Given $(V, E) \in \mathbf{T}(V, \mathcal{L}_\mathsf{E})$, an assignment $I : V \to \mathcal{L}_\mathsf{N}$ of node labels to nodes, and a total order $<$ on V, we say that

$G = (V, E, I, <)$ is a well-ordered tree if it satisfies the following conditions:

$$w{-}\ell_1{\rightarrow}_G\, w_1 \;\wedge\; w{-}\ell_2{\rightarrow}_G\, w_2 \;\wedge\; \ell_1 \prec \ell_2 \quad\Rightarrow\quad w_1 < w_2 \tag{9.1}$$
$$w_1 \rightarrow^*_G w'_1 \;\wedge\; w_2 \rightarrow^*_G w'_2 \;\wedge\; w_1 < w_2 \quad\Rightarrow\quad w'_1 < w'_2 \tag{9.2}$$
$$w{-}\ell_1{\rightarrow}_G\, w_1 \;\wedge\; I(w) = \ell_2 \;\wedge\; \ell_1 \prec \ell_2 \quad\Rightarrow\quad w_1 < w \tag{9.3}$$
$$w{-}\ell_1{\rightarrow}_G\, w_1 \;\wedge\; I(w) = \ell_2 \;\wedge\; \ell_2 \prec \ell_1 \quad\Rightarrow\quad w < w_1 \tag{9.4}$$

We define the additional functions $\mathsf{proj}^\ell_G : V \rightarrow 2^V$ for $\ell \in \mathcal{L}$:

$$\mathsf{proj}^\ell_G = \rightarrow^*_G \circ -\ell\rightarrow_G \qquad \text{for } \ell \in \mathcal{L}_\mathsf{E}$$
$$\mathsf{proj}^\ell_G(w) = \begin{cases} \{w\} & \text{if } \; I(w) = \ell \\ \emptyset & \text{otherwise} \end{cases} \qquad \text{for } \ell \in \mathcal{L}_\mathsf{N}$$

$\mathsf{proj}^\ell_G(w)$ for $\ell \in \mathcal{L}_\mathsf{E}$ is the set of nodes in the subtrees rooted at w's ℓ-daughters. We overload $<$ to obtain a partial order on 2^V as follows:

$$S_1 < S_2 \quad\equiv\quad \forall w_1 \in S_1,\ \forall w_2 \in S_2\ w_1 < w_2 \qquad \forall S_1, S_2 \subseteq V$$

It can be shown easily that the well-ordering conditions (9.1–9.4) are satisfied iff the following property holds:

$$\forall \ell_1, \ell_2 \in \mathcal{L},\ \forall w \in V \quad \ell_1 \prec \ell_2 \;\Rightarrow\; \mathsf{proj}^{\ell_1}_G(w) < \mathsf{proj}^{\ell_2}_G(w) \tag{9.5}$$

Thus the well-ordering conditions can be simply realized by sequentiality constraints between sets. The well-ordered labeled trees with nodes V, edge labels in $\mathcal{L}_\mathsf{E}$, node labels in $\mathcal{L}_\mathsf{N}$, and respecting the total order $\prec$ on $\mathcal{L}_\mathsf{E} \uplus \mathcal{L}_\mathsf{N}$ are in bijection with the solutions of $\mathbf{\Phi}_\mathsf{LP}(V, \mathcal{L}_\mathsf{E}, \mathcal{L}_\mathsf{N}, \prec)$ shown in Figure 10.

$$\begin{array}{l}
\mathbf{\Phi}_\mathsf{LP}(V, \mathcal{L}_\mathsf{E}, \mathcal{L}_\mathsf{N}, \prec) \;\equiv\; \mathbf{\Phi}_\mathsf{ID}(V, \mathcal{L}_\mathsf{E}) \wedge \\
\quad \forall w \in V \\
\qquad \{w\} = \uplus\{\ell(w) \mid \ell \in \mathcal{L}_\mathsf{N}\} \\
\quad \wedge\, \forall \ell \in \mathcal{L}_\mathsf{N} \quad \mathsf{proj}^\ell(w) = \ell(w) \;\wedge\; |\ell(w)| \neq 0 \;\equiv\; \ell = I(w) \\
\quad \wedge\, \forall \ell \in \mathcal{L}_\mathsf{E} \quad \mathsf{proj}^\ell(w) = \cup\{\mathsf{eqdown}(w') \mid w' \in \ell(w)\} \\
\quad \wedge \forall \ell_1, \ell_2 \in \mathcal{L}_\mathsf{E} \uplus \mathcal{L}_\mathsf{N} \quad \ell_1 \prec \ell_2 \;\Rightarrow\; \mathsf{proj}^{\ell_1}(w) < \mathsf{proj}^{\ell_2}(w)
\end{array}$$

FIG. 10. well-formedness condition for ordered labeled trees

It is possible, and in practice desirable, to improve propagation by formulating stronger constraints. For example, the property that all projections must be convex (i.e. intervals without holes) may be written:

$$\forall \ell \in \mathcal{L}_\mathsf{E},\ \forall w \in V \quad \mathsf{convex}(\mathsf{proj}^\ell(w)) \tag{9.6}$$

The declarative semantics of $\mathsf{convex}(S)$ is that for all $w_1, w_2 \in S$, if $w_1 < w_2$ then for all w such that $w_1 < w < w_2$, also $w \in S$.

10 Constraint-Based Dependency Parsing Revisited

In this section, we revisit the treatment of dependency parsing of [4] which we briefly outlined in Section 2. We show how to instantiate the framework developed in the preceding sections to obtain a mathematical characterization of admissible dependency syntax trees which also has a reading as a constraint program. Modulo issues of programming language syntax, the result is a parser. As in Section 2, we define a dependency grammar $\mathcal{G}$ by a 6-tuple:

$$\mathcal{G} = (\mathit{Strs}, \mathit{Cats}, \mathit{Agrs}, \mathit{Roles}, \mathit{Lexicon}, \mathit{Rules})$$

where $\mathit{Strs} = \{\texttt{"Buch"}, \texttt{"hat"}, \ldots\}$ is a finite set of strings notating the fully inflected forms of words, $\mathit{Cats} = \{\texttt{d}, \texttt{n}, \texttt{v\_fin}, \ldots\}$ is a finite set of categories which without loss of generality we can identify with (i.e. encode as) integers, $\mathit{Agrs} = \mathit{Gender} \times \mathit{Number} \times \mathit{Person} \times \mathit{Case} = \{\langle\texttt{masc sing 1 nom}\rangle, \ldots\}$ is a finite set of agreement tuples which again we can identify with integers, $\mathit{Roles} = \{\texttt{det}, \texttt{adj}, \texttt{subject}, \texttt{object}, \texttt{dative}, \ldots, \texttt{zuvinf}\}$ is a finite set of grammatical predicates to be used as edge labels. A lexical entry is an attribute value matrix with the following signature:

$$\begin{bmatrix} \mathsf{str} & : & \mathit{Strs} \\ \mathsf{cat} & : & \mathit{Cats} \\ \mathsf{agrs} & : & 2^{\mathit{Agrs}} \\ |\cdot|_{\texttt{det}} & : & 2^{\mathbb{N}} \\ & \vdots & \\ |\cdot|_{\texttt{zuvinf}} & : & 2^{\mathbb{N}} \end{bmatrix}$$

which specifies phonology, category, possible agreements, and valency restrictions. The *Lexicon* is a finite set of lexical entries, and we choose a function $\mathsf{lookup} : \mathit{Strs} \to \mathit{Lexicon}^*$ such that $\forall s \in \mathit{Strs}$, $\mathsf{lookup}(s)$ is a sequence without duplicates formed from the elements of $\{e \mid e \in \mathit{Lexicon} \wedge \mathsf{str}(e) = s\}$, i.e. such that:

$$e \in \mathsf{lookup}(s) \quad \equiv \quad \mathsf{str}(e) = s$$

An example lexical entry for ‘Buch’ is:

$$\begin{bmatrix} \mathsf{str} & : & \texttt{"Buch"} \\ \mathsf{cat} & : & \texttt{n} \\ \mathsf{agrs} & : & \{\langle\texttt{masc sing 3 nom}\rangle, \\ & & \;\langle\texttt{masc sing 3 acc}\rangle, \\ & & \;\langle\texttt{masc sing 3 dat}\rangle\} \\ |\cdot|_{\texttt{det}} & : & \{1\} \\ |\cdot|_{\texttt{adj}} & : & \{0, \ldots, \mu\} \\ |\cdot|_{\texttt{subject}} & : & \{0\} \\ & \vdots & \end{bmatrix}$$

It requires one determiner, permits any number of adjectives, forbids a subject, etc... Attribute agrs is identified as a disjunctive attribute and we write agr for $\mathsf{the}_{\mathsf{agrs}}$. Finally, *Rules* is a family (Γ_ℓ) of binary predicates indexed by roles and $\Gamma_\ell(w, w')$ must be satisfied to license a syntactic dependency labeled ℓ from head w to argument w'. Posing $\text{NP} = \{\mathtt{n}, \mathtt{pro}\}$ for the set of noun phrase categories and $\text{NOM} = \mathit{Gender} \times \mathit{Number} \times \mathit{Person} \times \{\mathtt{nom}\}$ for the set of agreements with case nominative, we define these predicates using the language of Section 7. For illustration, here are the predicates for roles `det` and `subject`:

$$\begin{array}{rcll}\Gamma_{\mathtt{det}} & \equiv & \lambda w, w' \cdot & \mathsf{cat}(w') = \mathtt{d} \ \wedge\ \mathsf{agr}(w) = \mathsf{agr}(w') \\ \Gamma_{\mathtt{subject}} & \equiv & \lambda w, w' \cdot & \mathsf{cat}(w') \in \text{NP} \ \wedge\ \mathsf{agr}(w) = \mathsf{agr}(w') \ \wedge\ \mathsf{agr}(w') \in \text{NOM}\end{array}$$

Constraint Formulation. We now provide the translation scheme which transforms an input sentence $s_1 \ldots s_n$ into a constraint $[\![s_1 \ldots s_n]\!]$ that precisely characterizes its valid dependency analyses according to our grammar $\mathcal{G}$. The translation follows the framework developed in the preceding sections. We introduce a set $V = \{w_1, \ldots, w_n\}$ of nodes, one for each word of the input sentence. We pose $\mathcal{L} = \mathit{Roles}$, $\mathcal{E} = \mathit{Lexicon}$, $\mathcal{A} = \{\mathsf{cat}, \mathsf{agrs}, |\cdot|_{\mathtt{det}}, \ldots, |\cdot|_{\mathtt{zuvinf}}\}$, $\mathcal{A}_d = \{\mathsf{agrs}\}$, and $\mathsf{lex}(w_i) = \mathsf{lookup}(s_i)$ for $1 \leq i \leq n$. All valid analyses are given by the solutions of:

$$\mathbf{\Phi}_{\mathsf{ID}}(V, \mathcal{L}) \ \wedge\ \mathbf{\Phi}_{\mathsf{LEX}}(V, \mathcal{E}, \mathcal{A}, \mathsf{lex}) \ \wedge\ \mathbf{\Phi}_{\mathsf{VAL}}(V, \mathcal{L}, \mathcal{E}, \mathcal{A}) \ \wedge\ \mathbf{\Phi}_{\mathsf{P}}(V, \mathcal{L}, \mathcal{E}, \mathcal{A}, (\Gamma_\ell))$$

thus we arrive at the translation shown in Figure 11.

In order to obtain a concrete parser, we must also stipulate a search strategy. In practice, the following strategy has proven quite satisfactory: first, apply distribution rules to determine all daughter sets $\ell(w)$ for $w \in V$, $\ell \in \mathcal{L}$, then apply distribution rules to determine all other attributes.

The account of dependency parsing presented here is only concerned with syntactic dependencies and ignores word-order. We have pursued two avenues of approach for the treatment of linear precedence:

Statistically Preferred Reading. The first approach for the treatment of word-order is based on statistical methods and was developed jointly with Thorsten Brants. Assuming the input sentence is a well-formed utterance, of all the analyses which our program is able to derive, some are more likely to correspond to the actual linearization than others. Thus, the idea is to inform the search strategy using statistics derived from a corpus. At each choice point where we need to decide an edge, we pick first the one which the statistical oracle ranks as most likely. This approach has proven quite successful at deriving first the intended reading, even in the presence of repeated extrapositions.

Topological Dependency Trees. The second approach aims at formalizing the principles of linear precedence and is the subject of current research with Ralph Debusmann. Corresponding to the non-ordered tree of syntactic dependencies, we postulate

$$
\begin{array}{l}
[\![s_1 \ldots s_n]\!] \;= \\
\qquad\qquad V = \mathsf{roots} \uplus \uplus\{\mathsf{daughters}(w) \mid w \in V\} \\
\qquad \wedge \quad |\mathsf{roots}| = 1 \\
\qquad \wedge \quad \forall w \in V \\
\qquad\qquad \begin{array}{rrcl}
 & \mathsf{eqdown}(w) & = & \{w\} \uplus \mathsf{down}(w) \\
\wedge & \mathsf{down}(w) & = & \cup\langle \mathsf{eqdown}(w_1), \ldots, \mathsf{eqdown}(w_n)\rangle[\mathsf{daughters}(w)] \\
\wedge & \mathsf{daughters}(w) & = & \mathtt{det}(w) \uplus \ldots \uplus \mathtt{zuvinf}(w) \\
\wedge & \mathsf{cat}(w) & = & \langle \mathsf{cat} \quad \mid \mathsf{lex}(w)\rangle[\mathsf{entry}(w)] \\
\wedge & \mathsf{agrs}(w) & = & \langle \mathsf{agrs} \quad \mid \mathsf{lex}(w)\rangle[\mathsf{entry}(w)] \\
\wedge & |w|_{\mathtt{det}} & = & \langle |\cdot|_{\mathtt{det}} \quad \mid \mathsf{lex}(w)\rangle[\mathsf{entry}(w)] \\
\vdots & & \vdots & \\
\wedge & |w|_{\mathtt{zuvinf}} & = & \langle |\cdot|_{\mathtt{zuvinf}} \mid \mathsf{lex}(w)\rangle[\mathsf{entry}(w)] \\
\wedge & |\mathtt{det}(w)| & \in & |w|_{\mathtt{det}} \\
\vdots & & \vdots & \\
\wedge & |\mathtt{zuvinf}(w)| & \in & |w|_{\mathtt{zuvinf}} \\
\wedge & \mathsf{agr}(w) & \in & \mathsf{agrs}(w) \\
\wedge & \forall w' \in V & & w' \in \mathtt{det}(w) \Rightarrow \Gamma_{\mathtt{det}}(w, w') \\
 & & & \qquad\qquad \vdots \\
 & & & \wedge\, w' \in \mathtt{zuvinf}(w) \Rightarrow \Gamma_{\mathtt{zuvinf}}(w, w')
\end{array}
\end{array}
$$

FIG. 11. Translation into constraint

a tree of topological dependencies (see Figure 9) that is partially ordered and projective. These two structures are mutually constraining. This approach instantiates the framework developed in the present article including the extension of Section 9 for projective partially ordered analyses. In [6], we used it to provide an elegant account of the challenging phenomena in the German verb complex.

11 Constraint Programming

In this section, we precisely describe the constraint programming support necessary and sufficient to express and solve the CSPs presented earlier in the article. We propose a formal system of constraints and inference rules, and define constraint propagation as deterministic inferential saturation.

Let $\Delta = \{0, \ldots, \mu\}$ be an interval of integers for some sufficiently large practical limit μ. We assume an infinite set of FD variables written I, I', I_j with values in Δ and an infinite set of FS variables written S, S', S_j with values in 2^{Δ}. We write D, D', D_j for a 'domain', i.e. a fixed subset of Δ, and i, j, k, n for particular integers in Δ.

$$
\begin{array}{lrcll}
\textsf{Initialization} & & & & \\
& & \rightarrow & I \in \Delta & \\
& & \rightarrow & \emptyset \subseteq S & \\
& & \rightarrow & S \subseteq \Delta & \\
\textsf{Clash} & & & & \\
& I \in \emptyset & \rightarrow & \textsf{false} & \\
& D_1 \subseteq S \wedge S \subseteq D_2 \wedge D_1 \not\subseteq D_2 & \rightarrow & \textsf{false} & \\
\textsf{Strengthen} & & & & \\
& I \in D_1 \wedge I \in D_2 & \rightarrow & I \in D_1 \cap D_2 & \\
& D_1 \subseteq S \wedge D_2 \subseteq S & \rightarrow & D_1 \cup D_2 \subseteq S & \\
& S \subseteq D_1 \wedge S \subseteq D_2 & \rightarrow & S \subseteq D_1 \cap D_2 & \\
\textsf{Weaken} & & & & \\
& I \in D & \rightarrow & I \in D' & D \subseteq D' \subseteq \Delta \\
& D \subseteq S & \rightarrow & D' \subseteq S & D' \subseteq D \\
& S \subseteq D & \rightarrow & S \subseteq D' & D \subseteq D' \subseteq \Delta
\end{array}
$$

FIG. 12. Rules for basic constraints

11.1 Basic Constraints

Information about a satisfying assignment will be refined incrementally by alternating steps of deterministic ‘propagation’ and non-deterministic ‘distribution’. Therefore we need means to represent partial information about an assignment and this is realized by ‘basic constraints’.

A basic constraint for a FD variable I takes the form $I \in D$ for a domain D. On the other hand, the assignment to a FS variable S is approximated by lower and upper bounds, i.e. by basic constraints $D_1 \subseteq S$ and $S \subseteq D_2$.

Basic constraints are given by the following abstract syntax:

$$B \quad ::= \quad \textsf{false} \;\mid\; I \in D \;\mid\; D \subseteq S \;\mid\; S \subseteq D \;\mid\; B_1 \wedge B_2$$

and they are subject to the inference rules of Figure 12. Saturation under these rules guarantees that a constraint B is either inconsistent (i.e. contains false) or that for each FD variable I in B, there is a most specific basic constraint $I \in D$, and that for each FS variable S in B, there are most specific lower- and upperbound basic constraints $D_1 \subseteq S$ and $S \subseteq D_2$. Posing:

$$
\begin{array}{rcl}
\lfloor I \rceil & = & \cap\{D \mid (I \in D) \in B\} \\
\lfloor S \rfloor & = & \cup\{D \mid (D \subseteq S) \in B\} \\
\lceil S \rceil & = & \cap\{D \mid (S \subseteq D) \in B\}
\end{array}
$$

when B is saturated, all of $I \in \lfloor I \rceil$, resp. $\lfloor S \rfloor \subseteq S$ and $S \subseteq \lceil S \rceil$ are in B and are the most specific bounds on variables I, resp. S. Indeed, we should use B as a subscript

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
i \in S & \equiv & \{i\} \subseteq S & S = D & \equiv & D \subseteq S \wedge S \subseteq D \\
i \notin S & \equiv & S \subseteq \Delta \setminus \{i\} & I \leq n & \equiv & I \in \{n, \ldots, \mu\} \\
I = i & \equiv & I \in \{i\} & n \leq I & \equiv & I \in \{0, \ldots, n\} \\
I \neq i & \equiv & I \in \Delta \setminus \{i\} & & &
\end{array}
$$

FIG. 13. Abbreviations

and write $\lfloor I \rceil_B$, $\lfloor S \rfloor_B$ and $\lceil S \rceil_B$, but, since this is never ambiguous, we will omit it to avoid notational clutter. All inference rules given in the following are motonously increasing in their premises: more specific premises yield more specific conclusions. For this reason, it is sufficient and simpler to express them in terms of $\lfloor I \rceil$, $\lfloor S \rfloor$ and $\lceil S \rceil$.

11.2 Non-Basic Constraints

We now extend our constraint language with 'non-basic' constraints, also known as 'propagators', and express their semantics in the form of inference rules.

$$
\begin{array}{rcl}
C & ::= & B \mid C_1 \wedge C_2 \mid I \in S \mid I \notin S \mid S = \{I\} \mid |S| = I \mid \\
& & I_1 \leq I_2 \mid S_1 \subseteq S_2 \mid S_1 \prec S_2 \mid \mathsf{convex}(S) \mid \\
& & S = \cup\langle S_1, \ldots, S_n \rangle[S'] \mid S = \cap\langle S_1, \ldots, S_n \rangle[S']
\end{array}
$$

To increase legibility, we adopt the abbreviations of Figure 13. For example the semantics of the membership constraint $I \in S$ is given by the following inference rules:

$$
\begin{array}{rcl}
& \to & I \in \lceil S \rceil \\
I = i & \to & i \in S
\end{array}
$$

Figure 14 lists all primitive binary constraints and Figure 15 our two primitive selection constraints. We can further extend our constraint language with the following derived constraints defined in Figure 16:

$$
\begin{array}{rcl}
C & ::= & \ldots \mid I_1 = I_2 \mid S_1 = S_2 \mid S_1 \parallel S_2 \mid \\
& & S = S_1 \cup \cdots \cup S_n \mid S = S_1 \cap \cdots \cap S_n \mid S_1 \prec \cdots \prec S_n \mid \\
& & S = \langle S_1, \ldots, S_n \rangle[I'] \mid I = \langle I_1, \ldots, I_n \rangle[I']
\end{array}
$$

The curious reader will find in [18] a detailed account of set constraints in Oz [19].

11.3 Disjunctive Propagators

Finally, we extend our constraint language with 'disjunctive' propagators:

$$
C \quad ::= \quad \ldots \mid C_1 \ \mathsf{or}\ C_2
$$

$$
\begin{array}{|llrcl|}
\hline
I \in S & \equiv & & & \\
& & & \rightarrow & I \in \lceil S \rceil \\
& & I = i & \rightarrow & i \in S \\
\hline
I \notin S & \equiv & & & \\
& & I = i & \rightarrow & i \notin S \\
& & & \rightarrow & I \in \Delta \setminus \lfloor S \rfloor \\
\hline
I_1 \leq I_2 & \equiv & & & \\
& & & \rightarrow & I_2 \in \{\min(\lfloor I_1 \rceil), \ldots, \mu\} \\
& & & \rightarrow & I_1 \in \{0, \ldots, \max(\lfloor I_2 \rceil)\} \\
\hline
S_1 \subseteq S_2 & \equiv & & & \\
& & & \rightarrow & \lfloor S_1 \rfloor \subseteq S_2 \\
& & & \rightarrow & S_1 \subseteq \lceil S_2 \rceil \\
\hline
|S| = I & \equiv & & & \\
& & & \rightarrow & |\lfloor S \rfloor| \leq I \\
& & & \rightarrow & I \leq |\lceil S \rceil| \\
& & n \leq I \wedge |\lceil S \rceil| = n & \rightarrow & \lceil S \rceil \subseteq S \\
& & I \leq n \wedge |\lfloor S \rfloor| = n & \rightarrow & S \subseteq \lfloor S \rfloor \\
\hline
S_1 \prec S_2 & \equiv & & & \\
& & \lfloor S_1 \rfloor \neq \emptyset & \rightarrow & S_2 \subseteq \{\min(\mu, \max(\lfloor S_1 \rfloor) + 1), \ldots, \mu\} \\
& & \lfloor S_2 \rfloor \neq \emptyset & \rightarrow & S_1 \subseteq \{0, \ldots, \max(0, \min(\lfloor S_2 \rfloor) - 1)\} \\
\hline
\mathsf{convex}(S) & \equiv & & & \\
& & \lfloor S \rfloor \neq \emptyset & \rightarrow & \{\min(\lfloor S \rfloor), \ldots, \max(\lfloor S \rfloor)\} \subseteq S \\
\hline
S = \{I\} & \equiv & & & \\
& & & & I \in S \ \wedge \ |S| = I' \ \wedge \ I' = 1 \\
& & & \rightarrow & S \subseteq \lfloor I \rceil \\
\hline
\end{array}
$$

FIG. 14. Constraints as sets of inference rules

The declarative semantics of C_1 or C_2 is simply that of disjunction. In Logic Programming, the only method for processing complex disjunctions is non-determinism. Thus in Prolog, writing C_1 ; C_2 operationally results in first trying C_1, and, if that fails, backtracking and trying C_2 instead. This has several drawbacks: (1) it is not sound (failure to prove C_1 is not the same as proving $\neg C_1$), (2) it forces the computation to commit immediately to exploring either one alternative or the other.

Early commitment is a poor strategy. It is often preferable to delay a choice until sufficient information is available to reject one of the alternatives. That is the intuition underlying the disjunctive propagator: C_1 or C_2 is a propagator not a choice point. It blocks until either C_1 or C_2 becomes inconsistent with respect to the current store of basic constraints: at that point, the propagator commits, i.e. reduces to, the remaining alternative. In this way, a disjunctive propagator has the declarative semantics of sound

$$
\boxed{\begin{array}{rcl}
S = \cup\langle S_1,\dots,S_n\rangle[S'] & \equiv & \\
 & \rightarrow & S' \subseteq \{1,\dots,n\} \\
 & \rightarrow & S \subseteq \cup\{\lceil S_j\rceil \mid j \in \lceil S'\rceil\} \\
 & \rightarrow & \cup\{\lfloor S_j\rfloor \mid j \in \lfloor S'\rfloor\} \subseteq S \\
\lfloor S_j\rfloor \not\subseteq \lceil S\rceil & \rightarrow & j \notin S' \\
\lfloor S\rfloor \setminus \cup\{\lceil S_j\rceil \mid j \in \lceil S'\rceil \setminus \{k\}\} \neq \emptyset & \rightarrow & \\
\multicolumn{3}{r}{k \in S' \wedge \lfloor S\rfloor \setminus \cup\{\lceil S_j\rceil \mid j \in \lceil S'\rceil \setminus \{k\}\} \subseteq S_k}
\end{array}}
$$

$$
\boxed{\begin{array}{rcl}
S = \cap\langle S_1,\dots,S_n\rangle[S'] & \equiv & \\
 & \rightarrow & S' \subseteq \{1,\dots,n\} \\
 & \rightarrow & \cap\{\lfloor S_j\rfloor \mid j \in \lceil S'\rceil\} \subseteq S \\
\lfloor S'\rfloor \neq \emptyset & \rightarrow & S \subseteq \cap\{\lceil S_j\rceil \mid j \in \lfloor S'\rfloor\} \\
\lfloor S\rfloor \not\subseteq \lceil S_j\rceil & \rightarrow & j \notin S' \\
\cap\{\lfloor S_j\rfloor \mid j \in \lfloor S'\rfloor \setminus \{k\}\} \setminus \lceil S\rceil \neq \emptyset & \rightarrow & \\
\multicolumn{3}{r}{k \in S' \wedge S_k \subseteq \Delta \setminus (\cap\{\lfloor S_j\rfloor \mid j \in \lfloor S'\rfloor \setminus \{k\}\} \setminus \lceil S\rceil)}
\end{array}}
$$

FIG. 15. Selection constraints rules

$$
\begin{array}{rcl}
I_1 = I_2 & \equiv & I_1 \leq I_2 \wedge I_2 \leq I_1 \\
S_1 = S_2 & \equiv & S_1 \subseteq S_2 \wedge S_2 \subseteq S_1 \\
S = S_1 \cup \cdots \cup S_n & \equiv & S = \cup\langle S_1,\dots,S_n\rangle[S'] \wedge S' = \{1,\dots,n\} \\
S = S_1 \cap \cdots \cap S_n & \equiv & S = \cap\langle S_1,\dots,S_n\rangle[S'] \wedge S' = \{1,\dots,n\} \\
S_1 \prec \cdots \prec S_n & \equiv & \bigwedge\{S_i \prec S_j \mid 1 \leq i < j \leq n\} \\
S_1 \parallel S_2 & \equiv & S = S_1 \cap S_2 \wedge S = \emptyset \\
S = \langle S_1,\dots,S_n\rangle[I'] & \equiv & S = \cup\langle S_1,\dots,S_n\rangle[S'] \wedge S' = \{I'\} \\
I = \langle I_1,\dots,I_n\rangle[I'] & \equiv & \\
\multicolumn{3}{r}{S = \cup\langle S_1,\dots,S_n\rangle[I'] \wedge S = \{I\} \wedge \bigwedge\{S_j = \{I_j\} \mid 1 \leq j \leq n\}}
\end{array}
$$

FIG. 16. Derived constraints

logical disjunction, unlike Prolog's ';' operator which implements merely negation-as-failure. The operational semantics are given by the rules below:

$$
\frac{B \wedge C_1 \quad \rightarrow^* \quad \mathsf{false}}{B \wedge (C_1 \;\mathsf{or}\; C_2) \quad \rightarrow \quad C_2}
\qquad\qquad
\frac{B \wedge C_2 \quad \rightarrow^* \quad \mathsf{false}}{B \wedge (C_1 \;\mathsf{or}\; C_2) \quad \rightarrow \quad C_1}
$$

where we write $C \rightarrow^* \mathsf{false}$ to mean that false is in the deterministic saturation of C under all propagation rules. In practice, disjunctive propagators are frequently useful for expressing implicational constraints such as (7.1):

$$
w' \in \ell(w) \;\Rightarrow\; \Gamma_\ell(w, w') \quad \equiv \quad w' \in \ell(w) \wedge \Gamma_\ell(w, w') \;\;\mathsf{or}\;\; w' \notin \ell(w)
$$

A precursor of disjunctive propagators was the idea of a 'deep guard', investigated for example in the Logic Programming language AKL [14]. Deep guards have since been generalized and subsumed by first class notions of encapsulated speculative computations, supported in the constraint programming language Oz [19] via 'computation spaces' [23].

While disjunctive propagators are extremely powerful, many implicational constraints do not require their full power and may be adequately expressed by reified constraints [16, 18] which are more widely supported.

11.4 Search By Propagation And Distribution

The operational semantics of a constraint C is given by a system of inference rules as described above. Propagation is defined as inferential saturation under this system of inference rules and we write C^* for the saturation of C. A FD variable I is said to be 'determined' in C^* when $\lfloor I \rceil_{C^*}$ is a singleton. A FS variable S is determined when $\lfloor S \rfloor_{C^*} = \lceil S \rceil_{C^*}$. When C^* does not contain false and all variables are determined, we have found a satisfying assignment β defined as follows for all variables I, S in C:

$$\{\beta(I)\} = \lfloor I \rceil_{C^*} \qquad\qquad \beta(S) = \lfloor S \rfloor_{C^*} \tag{11.1}$$

Constraint propagation alone may not be sufficient to determine all variables. In such an eventuality, it is necessary to perform a non-deterministic choice: this is what we call a 'distribution' step. We formalize this notion using distribution rules. A distribution rule has the form $\phi \rightarrow \psi_1 \vee \psi_2$ and non-deterministically infers either ψ_1 or ψ_2 when precondition ϕ is satisfied. We extend the system of inference rules for constraint C with distribution rules as shown below for all variables I, S in C:

$$\begin{array}{rcl} i \in \lfloor I \rceil & \rightarrow & i = I \ \vee \ i \neq I \\ i \in \lceil S \rceil \setminus \lfloor S \rfloor & \rightarrow & i \in S \ \vee \ i \notin S \end{array}$$

A saturation of C under both propagation and distribution rules either contains false or determines all variables.

THEOREM 11.1
C is satisfiable iff it has a consistent saturation.

All rules given are valid implications, therefore C is equivalent to the disjunction of its saturations. Consequently, if C is satisfiable, at least one of its saturations does not contain false (*Soundness*). Conversely, every consistent saturation defines an assignment β of values to variables as shown in (11.1). This assignment defines a model of C. We show this for the 'selection union' constraint whose declarative semantics is given by:

$$S = \cup\{S_1, \ldots, S_n\}[S'] \quad \equiv \quad S = \cup\{S_j \mid 1 \leq j \leq n,\ j \in S'\}$$

The first three propagations rules stipulating its operational semantics are:

$$\begin{array}{ll} \rightarrow & S' \subseteq \{1, \ldots, n\} \\ \rightarrow & S \subseteq \cup\{\lceil S_j \rceil \mid j \in \lceil S' \rceil\} \\ \rightarrow & \cup\{\lfloor S_j \rfloor \mid j \in \lfloor S' \rfloor\} \subseteq S \end{array}$$

Therefore in a consistent saturation we have:

$$\beta(S') \subseteq \{1, \ldots, n\} \qquad \beta(S) = \cup\{\beta(S_j) \mid j \in \beta(S')\}$$

which proves that $\beta \models S = \cup\{S_j \mid 1 \leq j \leq n,\ j \in S'\}$. We can proceed similarly for every constraint in our language and thus establish by induction that, if C has a consistent saturation, it is satisfiable (*Completeness*).

Search Strategies. The formal framework remains uncommitted as to when to apply a non-deterministic distribution rule and which one to choose. It is clear that in order to minimize search, distribution rules should be postponed as long as possible. Only when propagation has reached a fixed point should we consider applying a distribution rule.

Which distribution rule to choose is the province of a search strategy. A well-known traditional search strategy is 'first-fail': it chooses a non-determined variable with the smallest number of remaining possible values and enumerates its assignments. The intent is to try to keep the branching factor low in the search tree.

12 Conclusion

In this article, we developed a concise mathematical formalization of finite trees with labeled edges. We have argued that this formulation can be regarded as the specification of a constraint satisfaction problem and that the latter can be solved directly and efficiently with constraint programming, provided we empower our mathematical instruments with a computational reading which grants them the operational semantics of constraint propagators.

We entertained various refinements of our framework of interest to the computational linguist: lexical attributes and the treatment of lexical ambiguity, lexicalized valency constraints, disjunctive attributes and the improvement of lexical economy, the formulation of grammatical principles, and a framework for a class of partially ordered projective trees.

A novelty of our approach is the central importance given to finite sets and constraints expressed in terms of variables denoting finite sets. Selection constraints, which we first proposed in [4], allow to give concise mathematical expressions a direct computational reading and are the foundation for an effective treatment of ambiguity that takes full advantage of constraint propagation and constructive disjunction. An important contribution of this article is the identification of a family of aggregative selection constraints. In particular, the 'selection union' constraint is revealed as particularly expressive and versatile.

Finally we addressed the issue of efficient processing and characterized the services expected of a constraint programming system through a formal system of constraints and inference rules of propagation and distribution. We showed that the search for solutions of a constraint satisfaction problem could be defined as the derivation of consistent saturations. It is our hope that having provided precise formal blueprints of the constraint propagation mechanisms required will facilitate their adoption and integration in other systems.

The general framework presented here can be variously instantiated. It underlies both our treatment of immediate dependence [4] and of linear precedence [6] for parsing with a dependency grammar.

References

[1] Gunnar Bech. *Studien über das deutsche Verbum infinitum*. Munksgaard, Kopenhagen, 1955.

[2] Mehmet Dincbas, Pascal Van Hentenryck, Helmut Simonis, Abderrahamane Aggoun, Thomas Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems FGCS-88*, pages 693–702, Tokyo, Japan, December 1988.

[3] Jochen Dörre and Andreas Eisele. Feature logic with disjunctive unification. In *COLING-90*, volume 2, pages 100–105, 1990.

[4] Denys Duchier. Axiomatizing dependency parsing using set constraints. In *Sixth Meeting on Mathematics of Language*, pages 115–126, Orlando, Florida, July 1999.

[5] Denys Duchier. Set constraints in computational linguistics – solving tree descriptions. In *Workshop on Declarative Programming with Sets (DPS'99)*, pages 91–98, September 1999.

[6] Denys Duchier and Ralph Debusmann. Topological dependency trees: A constraint-based account of linear precedence. In *39th Annual Meeting of the Association for Computational Linguistics (ACL 2001)*, Toulouse, France, 9–11 July 2001.

[7] Denys Duchier and Claire Gardent. A constraint-based treatment of descriptions. In H.C. Bunt and E.G.C. Thijsse, editors, *Third International Workshop on Computational Semantics (IWCS-3)*, pages 71–85, Tilburg, NL, January 1999.

[8] Denys Duchier and Joachim Niehren. Dominance constraints with set operators. In *Proceedings of the First International Conference on Computational Logic (CL2000)*, LNCS. Springer, July 2000.

[9] Denys Duchier and Stefan Thater. Parsing with tree descriptions: a constraint-based approach. In *Sixth International Workshop on Natural Language Understanding and Logic Programming (NLULP'99)*, pages 17–32, Las Cruces, New Mexico, December 1999.

[10] Markus Egg, Joachim Niehren, Peter Ruhrberg, and Feiyu Xu. Constraints over lambda-structures in semantic underspecification. In *Joint Conf. COLING/ACL*, pages 353–359, 1998.

[11] Dale Gerdemann. *Parsing and Generation of Unification Grammars*. PhD thesis, University of Illinois, 1991.

[12] Carmen Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.

[13] John Griffith. Modularizing contexted constraints. In *COLING-96*, 1996.

[14] Sverker Janson. *AKL - A Multiparadigm Programming Language*. PhD thesis, SICS Swedish Institute of Computer Science, SICS Box 1263, S-164 28 Kista, Sweden, 1994. SICS Dissertation Series 14.

[15] Ronald M. Kaplan and Joan Bresnan. Lexical-functional grammar: A formal system for grammatical representation. In Joan Bresnan, editor, *The Mental Representation of Grammatical Relations*, pages 173–281. MIT Press, 1982.

[16] Kim Marriott and Peter J. Stuckey. *Programming with Constraints. An Introduction*. The MIT Press, Cambridge, MA, USA, 1998.

[17] John T. Maxwell and Ronald M. Kaplan. An overview of disjunctive constraint satisfaction. In *Proceedings of the International Workshop on Parsing Technologies*, pages 18–27, 1989.

[18] Tobias Müller. *Constraint Propagation in Mozart*. Doctoral dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany, 2001. In preparation.

[19] Mozart Consortium. The Mozart programming system, 1999. `http://www.mozart-oz.org/`.

[20] Tobias Müller and Martin Müller. Finite set constraints in Oz. In François Bry, Burkhard Freitag, and Dietmar Seipel, editors, *13. Workshop Logische Programmierung*, pages 104–115, Technische Universität München, 1997.

[21] Carl Pollard and Ivan A. Sag. *Information-Based Syntax and Semantics*, volume 1 of *CSLI Lecture Notes*. CSLI, 1987.

[22] Carl Pollard and Ivan A. Sag. *Head-driven Phrase Structure Grammar*. Studies in Contemporary Linguistics. The University of Chicago Press, 1994.

[23] Christian Schulte. *Programming Constraint Services*. Doctoral dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany, 2000. To appear in Lecture Notes in Artificial Intelligence, Springer-Verlag.

Received 20 May 2001