Constraint Programming in Computational Linguistics

Alexander Koller and Joachim Niehren
University of the Saarland, Saarbrücken, Germany

Constraint programming is a programming paradigm that was originally invented in computer science to deal with hard combinatorial problems. Recently, constraint programming has evolved into a technology which permits to solve hard industrial scheduling and optimization problems. We argue that existing constraint programming technology can be useful for applications in natural language processing. Some problems whose treatment with traditional methods requires great care to avoid combinatorial explosion of (potential) readings seem to be solvable in an efficient and elegant manner using constraint programming. We illustrate our claim by two recent examples, one from the area of underspecified semantics and one from parsing.

1 Introduction

This paper is an overview of techniques of modern constraint programming (CP) and their current applications in computational linguistics. We argue that CP can be the foundation for efficient, elegant approaches to notorious problems in natural language processing.

Constraint programming is a paradigm that emerged from logic programming in the mid-eighties (Jaffar and Lassez, 1987; Marriott and Stuckey, 1998). The aim of constraint programming is to provide a general platform on which (typically NP-hard) combinatoric problems, such as scheduling and optimization, can be solved efficiently. Traditionally, one would use a *generate and test* strategy, where a space of models is generated and searched for solutions; but the complexity and

[*]E-mail: `koller@coli.uni-sb.de` and `niehren@ps.uni-sb.de`. Special thanks to Denys Duchier, who developed much of the material we present. We are also grateful to our anonymous reviewers for helpful comments. The research reported here has been supported by the Collaborative Research Center (Sonderforschungsbereich) 378 of the DFG and the Esprit Working Group CCL II (EP 22457).

Submitted for
Proceedings of LLC8
D. Barker-Plummer, D. Beaver, J. van Benthem, P. Scotto di Luzio (eds.)

size of these problems makes the search spaces so huge that this strategy is unfeasible in practice. CP follows a strategy of *propagate and distribute* instead. The case distinctions that constitute search steps (distribution) are put off for as long as possible; first, simple deterministic inferences (propagation) are applied. That is, *model elimination* is preferred over *model enumeration*. Search will still be necessary, but good propagation will narrow down the possible choices in the distribution steps, thus reducing the size of the search space, sometimes dramatically.

In the nineties, constraint programming has evolved into a mature technology, and off-the-shelf development systems are available. CP has been incorporated into several host languages, e.g. C++ (ILOG, 1999) and Prolog (Dincbas et al., 1988; Aggoun and Beldiceanu, 1993; Aggoun et al., 1995). There are also special programming languages for CP, e.g. the concurrent CP language Oz (Smolka, 1995; Mozart Consortium, 1999) and others (Smolka, 1998; Caseau and Laburthe, 1996). We will use Oz in this paper. CP has been applied successfully to scheduling and optimization problems in industry (ILOG, 1999; Aggoun and Beldiceanu, 1993). The most useful algorithms known for constraint propagation apply to finite domain, finite set, and arithmetic constraints. Modern programming systems allow programmers to combine these algorithms at a high level of abstraction. (Notice that the word "constraint" in "constraint programming" has only a superficial relation to its usage in "constraint-based" theories e.g. of grammar, such as HPSG and LFG.)

Computational linguistics has only recently been discovered as an application of CP. Duchier (1999) presented an approach to parsing dependency grammar which is based on constraint programming with finite sets. Koller et al. (1998) applied constraint programming to semantic underspecification, and Duchier and Gardent (1999), to discourse. In all of these instances, model elimination techniques are used to cope with ambiguities which could otherwise cause serious combinatorial problems.

In this paper, we survey the state of the art of constraint programming in natural language processing. Our goal is to illustrate the power of constraint programming at successful examples, in the hopes of encouraging more computational linguists to apply CP for their own purposes.

In Section 2, we explain model elimination informally and show how it can be used in principle to deal with ambiguities. In Section 3, we discuss the basic ideas of Constraint Programming. This section also introduces some of the most important types of constraints, in particular finite domain, finite set, and selection constraints. In Sections 4 and 5, we go through examples from Section 2 in more detail and show how CP can help to solve the ambiguity problem. First, we implement dominance constraints, as used in a certain approach to scope underspecification. Then we present a parser for dependency grammar based on CP. We conclude and point to future work in Section 6.

Implementations of the systems described in Sections 4 and 5, including source codes, can be found on the WWW (Duchier et al., 1999).

2 Model Elimination

In this section, we take a closer (but still informal) look at model elimination. We first consider an example that shows how model elimination can prevent combinatorial explosion, and introduces the basic intuition. Then we turn to the problem of handling ambiguity in computational linguistics, where combinatorial explosion can be a problem too. In this context, model elimination is extremely close in spirit to *underspecification*. The section concludes with two linguistic examples – one from semantic underspecification and one from parsing. Details of these examples will be filled in in Sections 4 and 5.

2.1 An Example

Combinatorial puzzles are problems where a combination of values for a certain set of variables must be found which satisfies given constraints. An example is the following equation, where each letter stands for a distinct digit and leading digits are non-zero:

```
    SEND
+   MORE
= MONEY
```

Combinatorial puzzles are the simplest way to visualize the danger of *combinatorial explosion*. The example has eight variables, each of which has 10 possible values. This means that there are 10^8 potential models, only one of which really satisfies the equation. One way of solving the puzzle is to *generate* all combinations of values for the variables and then to *test* for each combination if it is a solution. A generate-and-test C implementation for the example takes about 20 seconds, but real-world applications typically involve hundreds of variables, and the number of possible combinations grows exponentially. So generate-and-test is unfeasible in practice – even for only 15 variables instead of eight, the generate-and-test program would run for over six years.

An alternative approach is to infer additional information about solutions first. In the example, it is easy to see that `M` must take the value 1, and `S` must be at least 8; more inferences are possible. That is, we *eliminate* models (i.e. all models where `M` is not 1) instead of enumerating them. The (simple, computationally cheap) inferences we perform are called *propagation*. When no further propagation is possible and we have not found a model yet, we must *distribute*, i.e. perform a case distinction for a variable value; but distribution is delayed for as long as possible. This strategy can reduce the size of the search space (and thus, the runtime), sometimes dramatically: A standard propagate-and-distribute implementation of the example using *finite domain constraints* takes a search space of 7 nodes!

2.2 Ambiguity and Underspecification

A similar danger of combinatorial explosion is inherent in the treatment of ambiguities in natural language processing: Multiple ambiguities present in the same sentence can make the total number of readings grow exponentially. This is a challenge because ambiguities are very common in natural language, for example as *attachment ambiguities* and *scope ambiguities*:

(1.1) John saw the man with the telescope.

(1.2) Every man loves a woman.

Attachment ambiguities (1.1) are syntactic ambiguities of where a given constituent, most commonly a PP, should be attached to the syntax tree. In the example, "with the telescope" could modify either "the man" or "saw". Scope ambiguities (1.2) are semantic ambiguities of the relative scope of different scope-bearing elements, such as quantifiers or negation. In the example, the quantifiers "every man" and "a woman" can take scope over each other in both ways.

Even worse, there are "ambiguities" that don't really contribute to the readings of a sentence at all. For example, the word "that" in the sentence *Mary likes that rabbit* could be either a demonstrative pronoun, a relative pronoun, or a complementizer according to the lexicon; this must be represented as an ambiguity in an early stage of parsing. However, this ambiguity goes away later because the relative pronoun and complementizer readings won't be able to participate in a parse; they would be reflected in a chart parser as unproductive edges. In processing, we should be especially careful not to get exponential blowups due to intermediate ambiguities that aren't even justified by an actual exponential number of readings of the entire sentence. We

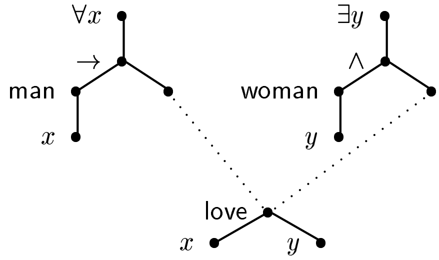

Figure 1.1: Underspecified description of "Every man loves a woman."

shall call the phenomenon of an ambiguity that is disambiguated (at least partially) by later processing steps *local ambiguity.*

Underspecification is an approach to ambiguity that attempts to cope with the problem of combinatorial explosion of readings. (For an overview of underspecification in semantics, see e.g. van Deemter and Peters, 1996.) The idea is to represent all the readings of an ambiguity with a single compact description and then to work with this description instead of the readings themselves for as long as possible. The readings can be obtained from the description, but this is only done when necessary; that is, underspecification prefers elimination of models over their enumeration.

For instance, a description of the two readings of (1.2) looks typically as in Fig. 1.1. The diagram specifies the parts that the semantics is composed of (the quantifiers "every man" and "a woman", and the nuclear scope "loves"), and that both quantifiers must outscope the nuclear scope. However, it leaves the relative order of the two quantifiers open. Note that as long as we don't enumerate readings explicitly, there is no room for combinatorial explosion: The size of the diagram is only linear in the size of any reading. It can be read more formally as a *dominance constraint*; we will come back to this in Section 4.

2.3 Example: Scope and Anaphora

An underspecified, model-eliminative approach to ambiguity makes it possible to deal with local ambiguity in an elegant way. By way of example, suppose we continue the ambiguous (1.2) with

(1.3) Her name is Mary.

This continuation forces the existential quantifier to take wide scope; the ambiguity goes away. One way to explain this is to assume a dynamic logic (e.g. Kamp and Reyle, 1993; Groenendijk and Stokhof, 1991) as our semantic representation language; in such a logic, the reading in which the universal quantifier gets wide scope would violate anaphoric accessibility conditions.

Clearly, a generate-and-test approach that enumerates all readings of the first sentence, then tests each reading for accessibility violations, will cause unacceptable processing times for more complex ambiguities, besides being not very reasonable intuitively.

An underspecified approach could represent the meaning of the first sentence e.g. as in Fig. 1.1. When the second sentence is processed, it hasn't yet committed to the actual meaning of the first sentence. We could derive from the anaphor the restriction that no static connective (such as a universal quantifier) should be on top of the existential quantifier and add this restriction to the description of the first sentence. Finally, we would use some calculus for making descriptions more explicit to derive the fact that the existential quantifier actually must take wide scope in the first sentence.

Thus, anaphora give us an opportunity for eliminating some models of a local ambiguity without ever representing them explicitly. For details of this construction, see (Koller and Niehren, 2000). In Section 4, we will come back to scope underspecification; there we will show how model-elimination techniques can be applied to make the *enumeration* of readings more feasible – as in the "Send More Money" puzzle above.

2.4 Example: Parsing

Finally, recall the earlier "that" example where the lexical entry and/or morphological form of a word could be ambiguous to a parser. Here we consider a similar example: the morphological ambiguity of the NP "die Frau" ("the woman") in German. This NP can be either in the nominative or in the accusative case. This is interesting in parsing a sentence like

(1.4)	*Den*	*Mann*	*liebt*	*die*	*Frau.*
	the(acc)	man	loves	the(nom/acc)	woman
	'The woman loves the man.'				

This sentence is not ambiguous; "die Frau" is the subject, and "den Mann" is the direct object. But German word order is relatively free, so a parser must also allow for "die Frau" being the object, as in

(1.5)	*Der*	*Mann*	*liebt*	*die*	*Frau.*
	the(nom)	man	loves	the(nom/acc)	woman

'The man loves the woman.'

In each case, a chart parser will probably introduce an edge which doesn't contribute to the correct parse.

Again, model elimination can point to a more intuitive treatment of a local ambiguity. We start with a description of the syntax specifying that "liebt" is the verb, that it has exactly one subject and one object, and what the available case information is. Now we can reason as follows: In the first sentence, "den Mann" must be the object because its case can only be accusative. So "die Frau" must be the subject because the subject role is already filled. The second example works in the same way.

We will develop a dependency parser based on Constraint Programming in Section 5 which will reduce the size of the search space by performing inferences just like these.

3 Constraint Programming

In this section, we make our notion of Constraint Programming more precise. After presenting the computation model of *concurrent constraint programming* at an example, we define a concrete constraint language. We will use this constraint language in the next two sections to treat the linguistic examples from Section 2. We can only scrape at the surface of CP here; the recommended introductory textbook is (Marriott and Stuckey, 1998).

3.1 An Example

On an abstract level, propagate and distribute is best explained by the model of *concurrent constraint programming* (Saraswat and Rinard, 1990; Saraswat et al., 1991). The idea is to distinguish *simple* constraints that can be solved deterministically from *complex* constraints. Simple constraints are stored in a *constraint store*, and complex constraints are turned into *propagators*, concurrent processes that observe the constraint store, deduce logical consequences from the store and their defining formulas, and add these consequence to the constraint store.

By way of example, let us look at how propagation works for *finite set constraints* (Fig. 1.2). The constraint store (oval) contains

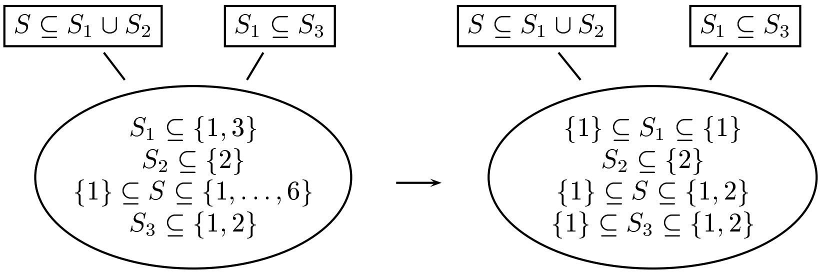

Figure 1.2: Propagation with set constraints.

(ground) lower and upper bounds for some set variables. Two propagators (rectangles) operate on the constraint store; they implement set inclusions. In the situation on the left hand side, both propagators can contribute information. First, the store entails that $S_1 \cup S_2 \subseteq \{1,2,3\}$, so the left propagator, $S \subseteq S_1 \cup S_2$, can infer that $4 \notin S \wedge 5 \notin S \wedge 6 \notin S$; this information is added to the store. Second, since the store entails $1 \in S \wedge 1 \notin S_2$, the left propagator can deduce $1 \in S_1$. On the other hand, the right propagator, $S_1 \subseteq S_3$, can infer from $3 \notin S_3$ that $3 \notin S_1$; and from $1 \in S_1$, which was added by the left propagator, it can infer $1 \in S_3$. But now, the store entails $S_1 \cup S_2 \subseteq \{1,2\}$, so the left propagator can again infer that $3 \notin S$. Now no further propagation is possible; the end result is shown on the right hand side of Fig. 1.2.

If we had to find a solution to the set constraint, we would now perform a single distribution step, which could e.g. create two new constraint stores, one with the information $S_2 = \emptyset$ and one with $S_2 = \{2\}$. Then propagation would resume over these two stores. But propagation had a tremendous effect already: We have fully determined the value for S_1, and there are only eight possible combinations of variable values left (instead of 1024).

3.2 A Constraint System

The concrete constraint system we are going to use for our applications is shown in Fig. 1.3. It defines several types of constraints C and several types of variables V. Some of these constraints are standard in constraint programming, such as finite domain (FD) constraints A, finite set (FS) constraints B, and tuple constraints F; some, such as disjunctive propagators D and selection constraints E, are less common. All of these constraints are fully supported by the Mozart programming

$$
\begin{array}{lcl}
V & ::= & S \mid X \mid N \\
A & ::= & T \in \{n_1, \dots, n_k\} \\
  & \mid & T_1 = T_2 \mid T_1 \neq T_2 \\
  & \mid & T_1 \leq T_2 \\
B & ::= & n \in S \mid n \notin S \\
  & \mid & S \subseteq \cup_{i=1}^{n} S_i \\
  & \mid & S_1 \cap S_2 = \emptyset
\end{array}
\qquad
\begin{array}{lcl}
C & ::= & A \mid B \mid D \mid E \mid F \\
D & ::= & \mathsf{or}_{i=1}^{n} \wedge_{j=1}^{m(i)} C_{ij} \\
T & ::= & n \mid N \mid |S| \\
  & \mid & \min(S) \mid \max(S) \\
E & ::= & S = \langle S_1, \dots, S_n \rangle [N] \\
F & ::= & X = [V_1, \dots, V_k] \\
  & \mid & X = Y
\end{array}
$$

Figure 1.3: The constraint system.

system, an implementation of the concurrent constraint language Oz (Mozart Consortium, 1999).

Variables and Values. We assume three infinite sets of *variables*, collectively ranged over by V. Let max be a large natural number depending on the constraint programming system in use, for instance $\mathsf{max} = 134217726$ for Oz. A *finite domain variable* N denotes a natural number n in $\{0, \dots, \mathsf{max}\}$; a *set variable* S denotes a finite subset of $\{0, \dots, \mathsf{max}\}$, and a tuple variable X denotes a k-tuple of values.

Constraint Store. The simple constraints that we can write directly into the constraint store are (conjunctions of) finite domain membership $N \in \{n_1, \dots, n_n\}$, set inclusion $n \in S$, set exclusion $n \notin S$, and tuple constraints. Set inclusion and exclusion describe lower and upper bounds of set variables S. For any set D of numbers, we write $D \subseteq S$ as a shortcut for $\wedge\{n \in S \mid n \in D\}$ and $S \subseteq D$ instead of $\wedge\{n \notin S \mid n \notin D\}$. All other constraints serve as propagators; we describe their behaviour below.

Distribution. Distribution performs case distinctions during the search for a solution. In this paper, we need distribution over the values of finite domain and finite set variables. FD distribution steps (shown in Fig. 1.4) split the domain $\{n_1, \dots, n_k\}$ of N into two subdomains $\{n_1, \dots, n_i\}$ and $\{n_{i+1}, \dots, n_k\}$, for some $1 \leq i < k$. Then the two cases $N \in \{n_1, \dots, n_i\}$ and $N \in \{n_{i+1}, \dots, n_k\}$ are considered independently. FS distribution distinguishes possible values for finite sets in a similar fashion. The constraint store and all propagators are copied by distribution. A *distribution strategy*, which determines which finite domains are split and in which way, can be specified independently.

Finite Set Constraints (B). Beyond the simple constraints for set inclusion and exclusion, the system provides union and disjointness constraints. These constraints are implemented as propagators with the following operational semantics:

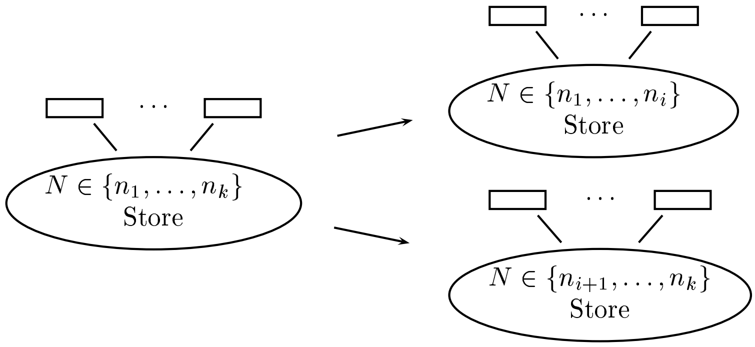

Figure 1.4: Distribution (where $1 \leq i < k$).

$S \subseteq \cup_{i=1}^{k} S_i$:	$\wedge_{i=1}^{k} n \notin S_i$	$\Rightarrow$	$n \notin S$	
$S \subseteq \cup_{i=1}^{k} S_i$:	$n \in S \wedge \wedge_{1 \leq i \neq j}^{k} n \notin S_i$	$\Rightarrow$	$n \in S_j$	if $1 \leq j \leq k$
$S_1 \cap S_2 = \emptyset$:	$n \in S_i$	$\Rightarrow$	$n \notin S_j$	if $i \neq j \in \{1, 2\}$

Each propagator waits until the constraint store contains the simple constraints on the left hand side of the rule. Then the simple constraint on the right hand side is added to the constraint store.

In the rest of the paper, we will write $S = \uplus_{i=1}^{n} S_i$ as a shortcut for the conjunction of $S \subseteq \cup_{i=1}^{n} S_i$, $\bigwedge_{i=1}^{n} S_i \subseteq S$, and $\bigwedge_{1 \leq i < j \leq n} S_i \cap S_j = \emptyset$; that is, $\uplus$ defines a partition of a set variable.

Finite Domain Constraints (A). In addition to the simple domain membership constraint, the system defines equality, inequality, and ordering constraints for finite domain (FD) variables N. In this paper, we use FD variables only as auxiliary tools, e.g. for reasoning about the cardinality of sets, controlling disjunctive propagators, or modeling agreement. Usually, FD constraints come together with arithmetic constraints, but we do not need those here.

Tuple Constraints (F). A tuple constraint $X = [V_1, \ldots, V_k]$ states that X denotes the tuple of denotations of $V_1, \ldots, V_k$. Tuple constraints can be stored directly in a constraint store, but may have to be synchronized with existing information by unification. For instance, if $X = [V_1, \ldots, V_k]$ is in the store, then the addition of $X = [V_1', \ldots, V_k']$ amounts to adding $\wedge_{i=1}^{k} V_i = V_i'$.

Disjunctive Propagators (D). Disjunctive propagators (Schulte, 2000; Janson and Haridi, 1991) are maybe the most unfamiliar type of constraint we define. On the level of denotation, the disjunctive propagator $\mathsf{or}_{i=1}^{n} \wedge_{j=1}^{m(i)} C_{ij}$ is equivalent to a disjunction $\vee_{i=1}^{n} \wedge_{j=1}^{m(i)} C_{ij}$. Operationally, a disjunctive propagator tests the n clauses in parallel for consistency with the global constraint store. It blocks until all but one clause have become inconsistent, and then it adds the remaining clause to the constraint store. That is, a disjunctive propagator *never* enumerates cases. This is in contrast to the behaviour of traditional logic programming languages (see e.g. Apt, 1990), where disjunction is modeled by choice points and *always* enumerated. Enumeration can be implemented in CP by distributing over *choice variables*, a technique which is explained and used in Section 4.

An example should help to make this clear. In Section 3.1, we had two propagators for set constraints. Now let us add a third propagator: the disjunctive propagator $\mathsf{or}(3 \in S, 2 \in S_2)$. The new propagator will block during all of the propagation we described earlier, as both branches are consistent with the constraint store. The last propagation step, however, infers that $3 \notin S$; so the first branch of the disjunctive propagator becomes inconsistent, and it can add the second branch $(2 \in S_2)$ to the constraint store.

It is possible to implement implications and equivalences using or if the antecedent is a constraint whose negation we can write down (e.g. set membership). We write $\mathsf{imply}(C,\ C')$ as an abbreviation of $\mathsf{or}(C \wedge C', \neg C)$ and $\mathsf{equiv}(C,\ C')$ for $\mathsf{or}(C \wedge C', \neg C \wedge \neg C')$. Note that these implementations allow propagation in both directions: If C is entailed by the constraint store, the propagator for $\mathsf{imply}(C,\ C')$ will add C' to the store, and if C' is inconsistent with the store, the propagator will add $\neg C$.

Selection Constraints. A different approach to disjunction is taken by selection constraints. The selection constraint $S{=}\langle S_1, \ldots, S_n\rangle[N]$ is logically equivalent to $\vee_{i=1}^{n}(N{=}i \wedge S{=}S_i)$; that is, the variable N selects one of the set variables S_i in the list. Selection constraints sacrifice generality for even stronger propagation: They provide *constructive disjunction*, which can extract information common to all alternatives.

$$
\begin{array}{lll}
S{=}\langle S_1, \ldots, S_m\rangle[N]: & N{\in}\{n_1, \ldots, n_k\} \wedge \wedge_{i=1}^{k} n{\in}S_{n_i} & \Rightarrow \quad n{\in}S \\
S{=}\langle S_1, \ldots, S_m\rangle[N]: & N{\in}\{n_1, \ldots, n_k\} \wedge \wedge_{i=1}^{k} n{\notin}S_{n_i} & \Rightarrow \quad n{\notin}S
\end{array}
$$

Two selection constraints can communicate with each other by sharing their selection variables. This is possible because selection variables N can be constrained by propagation:

$$S{=}\langle S_1, \ldots, S_n\rangle[N] : n{\in}S \wedge n{\notin}S_i \quad \Rightarrow \quad N{\neq}i$$

4 Application: Dominance Constraints for Scope Underspecification

In this section, we define *dominance constraints* and show how to encode them using finite set and finite domain constraints. This is a standard move in CP implementations: We reduce a language of *application-oriented* constraints to *implementation-oriented* constraints, which can be processed efficiently and for which propagation is automatically provided by the programming system.

This gives us an immediate way to *solve* dominance constraints: We simply have to solve the corresponding FS constraints. However, the propagation available in the encoding is useful even without distribution; Koller and Niehren (2000) show how the implementation below can be extended to do inferences as in Section 2.3 by pure propagation.

The presentation is based on (Koller and Niehren, 1999; Duchier and Gardent, 1999; Duchier and Niehren, 2000).

4.1 Dominance constraints

Dominance constraints are tree descriptions, i.e. logical formulas whose models are trees. They are omnipresent in underspecified approaches to scope ambiguity, sometimes in disguise (Reyle, 1993; Bos, 1996); they appear explicitly in (Muskens, 1995) and as part of the Constraint Language for Lambda Structures (CLLS, Egg et al., 1998). Dominance constraints have been applied to many other areas of computational linguists as well, e.g. incremental parsing (Marcus et al., 1983), tree adjoining grammars (Vijay-Shanker, 1992; Rambow et al., 1995), and discourse (Gardent and Webber, 1998); their logical and computational properties have been investigated e.g. in (Cornell, 1994; Backofen et al., 1995; Koller et al., 1998).

We assume a set of labels ranged over by f which contains *every*, *a*, *man*, *loves*, *woman*, etc. A dominance constraint φ is a conjunction of *labeling constraints* $X{:}f(X_1, \ldots, X_n)$, (atomic) *dominance constraints* $X \lhd^* Y$, *inequality constraints* $X \neq Y$, and *disjointness constraints* $X \perp Y$.

$$\varphi \quad ::= \quad X{:}f(X_1, \ldots, X_n) \mid X \lhd^* Y \mid X \perp Y \mid X \neq Y \mid \varphi \wedge \varphi'$$

The variables X, Y, X_i in a constraint denote nodes in the same tree. A labeling constraint expresses that X denotes a node which has the label f and whose children are denoted by $X_1, \ldots, X_n$. A dominance constraint $X \lhd^* Y$ expresses that X denotes a node that is somewhere above (or equal to) the node denoted by Y in the tree. An inequality

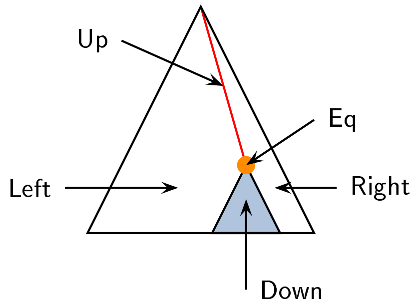

Figure 1.5: Partitioning Trees

constraint expresses that the two variables denote different nodes, and a disjointness constraint $X \perp Y$ expresses that neither of the two nodes dominates the other. We write $X{=}Y$ (equality) for the conjunction $X \lhd^* Y \wedge Y \lhd^* X$ and $X \lhd^+ Y$ (strict dominance) for $X \lhd^* Y \wedge X \neq Y$.

By way of illustration of the use of dominance constraints in CL, we sketch their application to scope underspecification (Egg et al., 1998). We can encode the terms and formulas of a traditional semantic representation language (such as formulas of predicate logic, or lambda terms) as trees; then we can take a dominance constraint φ to describe those formulas or terms whose corresponding trees satisfy φ. So we can use dominance constraints to give intuitive pictures such as Fig. 1.1 a precise meaning: We can read the graph as a graphical representation of a dominance constraint – a *constraint graph*. The nodes of the constraint graph represent variables in a dominance constraint; labels and solid lines represent labeling constraints, and dotted lines represent atomic dominance constraints between the respective variables. For more details, see (Egg et al., 1998, 2000).

4.2 A Solver for Dominance Constaints

Now we present the encoding of dominance constraints as set constraints. Solutions of these set constraints encode solutions of the dominance constraint, which represent the readings of the sentence.

The key idea for the encoding is to model a tree by partitioning the set of its nodes. Each node in a tree induces a partition of the set of variables: The node denoted by a variable X is either strictly above, strictly below, equal, or disjoint (to the left or to the right) of this node. The situation is illustrated by Figure 1.5. In logical terms, this means that the disjunction (1.6) is valid for all tree models:

(1.6) $\quad X{=}Y \vee X \triangleleft^+ Y \vee Y \triangleleft^+ X \vee X \perp Y$

Formula 1.6 suggests a naive generate and test procedure: In order to solve a dominance constraint φ, simply test all possible relationships between each two variables $X, Y \in \mathit{Var}(\varphi)$ for consistency. The set constraint encoding will be much more efficient, however, because of propagation on sets.

Translation to Set Constraints

Suppose we are given a dominance constraint φ. We associate each variable X that appears in φ with a unique natural number $\mathit{Id}(X)$. Let Ids be the set of all variable identities $\{\mathit{Id}(X) \mid X \in \mathit{Var}(\varphi)\}$. For all $X \in \mathit{Var}(\varphi)$, we introduce set variables $\mathit{eq}(X)$, $\mathit{down}(X)$, $\mathit{up}(X)$, and $\mathit{disj}(X)$, which we are going to constrain to the sets of variable identities equal to, strictly above, strictly below, or disjoint to X. For all $X \in \mathit{Var}(\varphi)$, we require:

$$\begin{array}{l} \mathit{Id}(X) \in \mathit{eq}(X) \wedge \\ \mathit{Ids} = \mathit{eq}(X) \uplus \mathit{down}(X) \uplus \mathit{up}(X) \uplus \mathit{disj}(X) \end{array}$$

We assume a tuple variable $\mathit{daughters}(X)$ and an integer variable $\mathit{label}(X)$, which represents the label f of the node denoted by X, encoded as some number. For each node variable $X \in \mathit{Var}(\varphi)$, we assume a tuple variable X of the same name and impose the tuple constraint

$$X = [\mathit{eq}(X), \mathit{up}(X), \mathit{disj}(X), \mathit{down}(X), \mathit{daughters}(X), \mathit{label}(X)]$$

We can use the tuple constraint $X = Y$ to express that two variables denote the same node. Finally, we introduce two auxiliary set variables $\mathit{eqdown}(X)$ and $\mathit{equp}(X)$ for all $X \in \mathit{Var}(\varphi)$ for which we impose the following constraints.

$$\begin{array}{rcl} \mathit{eqdown}(X) & = & \mathit{eq}(X) \uplus \mathit{down}(X) \\ \mathit{equp}(X) & = & \mathit{eq}(X) \uplus \mathit{up}(X). \end{array}$$

If X dominates Y in a tree, this means that X is above Y, Y is below X, and whatever is disjoint to X is also disjoint to Y. Thus:

$$\begin{array}{rcccc} [\![X \triangleleft^* Y]\!] & =_{def} & \mathit{equp}(X) & \subseteq & \mathit{equp}(Y) \\ & \wedge & \mathit{eqdown}(X) & \supseteq & \mathit{eqdown}(Y) \\ & \wedge & \mathit{disj}(X) & \subseteq & \mathit{disj}(Y). \end{array}$$

An encoding of the labeling constraint $X{:}f(X_1, \ldots, X_n)$ will obviously require that $X_1, \ldots, X_n$ denote the daughters of X and that X

has the label f. Furthermore, the down set of X is the disjoint union of the *eqdown* sets of the daughters, and the up set of each daugher is the *equp* set of X:

$$\begin{array}{rcrcl} [\![X{:}f(X_1,\ldots,X_n)]\!] & =_{def} & \mathit{label}(X) & = & f \\ & \wedge & \mathit{daughters}(X) & = & [X_1,\ldots,X_n] \\ & \wedge & \mathit{down}(X) & = & \uplus_{i=1}^n \mathit{eqdown}(X_i) \\ & \wedge & \wedge_{i=1}^n \mathit{up}(X_i) & = & \mathit{equp}(X) \end{array}$$

Inequality $X \neq Y$ simply means that the equal sets of the two variables are disjoint – that is,

$$[\![X \neq Y]\!] =_{def} \ \mathit{eq}(X) \cap \mathit{eq}(Y) = \emptyset.$$

Finally, disjointness $X \perp Y$ is encoded as

$$\begin{array}{rcrcl} [\![X \perp Y]\!] & =_{def} & \mathit{eqdown}(X) & \subseteq & \mathit{disj}(Y) \\ & \wedge & \mathit{eqdown}(Y) & \subseteq & \mathit{disj}(X). \end{array}$$

We can now implement the disjunctions (1.6) as disjunctive propagators (1.7), as provided by our constraint programming system.

$$\text{(1.7)} \qquad \begin{array}{l} N \in \{1,2,3,4\} \wedge \mathsf{or}(N{=}1 \wedge X{=}Y,\ N{=}2 \wedge [\![X \lhd^+ Y]\!], \\ \qquad\qquad N{=}3 \wedge [\![Y \lhd^+ X]\!],\ N{=}4 \wedge [\![X \perp Y]\!]) \end{array}$$

These disjunctive propagators enforce that every model of the constraint really encodes a tree.

Enumerating Solutions

The constraints so far allow us to axiomatize the encodings of trees satisfying a certain dominance constraint. Unfortunately, propagation alone will not be sufficient to actually enumerate these encodings. For instance, the propagator (1.7) can easily block because two different branches are consistent with all the other available information.

To solve this problem, we have introduced constraints involving finite domain variables N into the propagator. We introduce one such *choice variable* for each pair of node variables in φ and associate every clause of the corresponding disjunction with one possible value for N. Now when evaluation of an or statement suspends, we can distribute over the remaining values of its associated choice variable. Thus, choice variables allow us to control the enumeration of solutions.

We can do even better than this, however, if we split the single four-way disjunction into four two-way disjunctions:

Quantifiers	Readings	(a)	(b)
3	5	200 ms/6 failure	100 ms/0 failure
5	42	2.69 s/78 failure	2.63 s/0 failure
7	168	5.98 s/229 failure	6.24 s/0 failure

Figure 1.6: Runtimes of the dominance constraint solver for some problem sizes (a) using the four-way, (b) using four two-way or propagators.

$$\begin{array}{llll} & N \in \{1,2,3,4\} & \wedge & \mathsf{or}(N{=}1 \wedge X{=}Y,\ N{\neq}1 \wedge [\![X{\neq}Y]\!]) \\ (1.8) & & \wedge & \mathsf{or}(N{=}2 \wedge [\![X \lhd^+ Y]\!],\ N{\neq}2 \wedge [\![\neg X \lhd^+ Y]\!]) \\ & & \wedge & \mathsf{or}(N{=}3 \wedge [\![Y \lhd^+ X]\!],\ N{\neq}3 \wedge [\![\neg Y \lhd^+ X]\!]) \\ & & \wedge & \mathsf{or}(N{=}4 \wedge [\![Y \perp X]\!], N{\neq}4 \wedge [\![\neg Y \perp X]\!]) \end{array}$$

The declarative semantics of (1.7) and (1.8) are the same, but (1.8) propagates more strongly. The operational semantics of the disjunctive propagators is to block until all but one alternative are inconsistent; and it's clearly easier to make a single alternative inconsistent than three.

Results The constraints so far can be taken over almost verbatim as a real Oz program for enumerating solutions of a dominance constraint. All that's missing is an input procedure that takes a dominance constraint φ as defined in Section 4.1 and spawns the treeness propagators (1.8) and, for each atomic constraint in φ, the corresponding set constraint. Now we only have to distribute over the choice variables.

Running this implementation on many examples exhibits quite reasonable runtimes for enumerating solutions. Times for three examples with three, five, and seven scope-bearing elements (quantifiers, negation, sentence-embedding verbs, etc.) are shown in Fig. 1.6.[1]

It is interesting to observe that the propagators (1.8) are indeed much stronger than (1.7): In fact, they restrict the search space so strongly that there is *no* failure at all! This is also illustrated by Fig. 1.7, which shows the search tree for sentence (iii): All the leaves are (originally green) diamonds indicating success, and there are no (originally red) boxes indicating failure. However, the *run times* of the two implementations don't differ significantly; there is a more general trade-off between *strong* propagators and *cheap* propagators: What is

[1]The sentences were (i) *Two researchers of every company work on a program*, (ii) *Some researchers of every departement of most companies see most samples of every product*, (iii) *Every mother says most researchers of a company do not see every sample of a professor.*

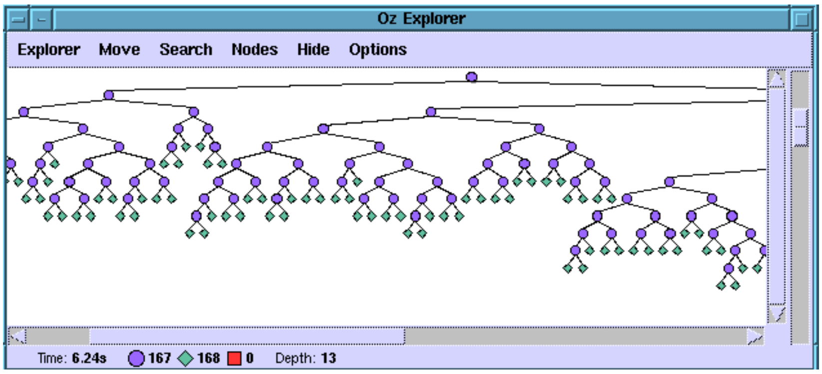

Figure 1.7: A failure free search tree.

gained by making the search tree small can be lost by doing more work in each of its nodes.

4.3 Discussion

What we have just done is to encode trees using tuples of sets and dominance constraints as finite set constraints. This was useful because there are implementations of set constraints with very good propagation (which, incidentally, are even supported on a high level of abstraction by some programming languages). By distribution, we could enumerate all models; but as we have indicated above, propagation alone can be useful.

Propagation really made a difference in avoiding combinatorial explosion. The stronger propagators could even avoid blind search altogether in the examples in Fig. 1.6; distribution was only necessary to distinguish different readings. This happens for all examples from scope underspecification that we have tried. It is a very surprising result because the satisfiability problem of dominance constraints is NP-complete (Koller et al., 1998), and it is characteristic of NP-complete problems that there must be *some* failure. Recently, it has become clear that the dominance constraints that are really needed for scope are all in a subclass called *normal*; normal dominance constraints were shown to have a polynomial satisfiability problem by Koller et al. (2000). This does not say anything about the behaviour of the CP implementation; but it seems that the propagators (1.8) are so strong that they exploit the polynomial complexity automatically.

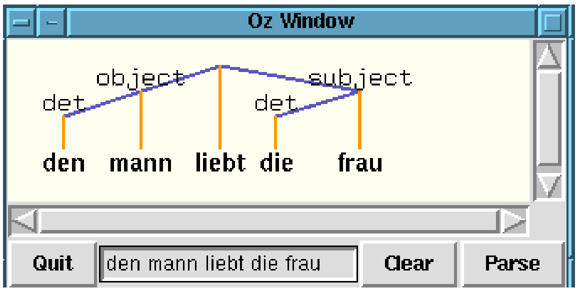

Figure 1.8: Dependency tree of *Den Mann liebt die Frau.*

5 Application: Dependency Parsing

In this section, we show how we can apply CP to dependency parsing, following (Duchier, 1999). After a brief introduction to the precise flavour of dependency grammar we use and a definition of a tiny grammar of German, we show how to axiomatize the valid dependency trees of a sentence by finite set constraints. Parsing, then, reduces to solving these constraints. The resulting implementation handles many inferences by constraint propagation: For instance, we show in detail how the parser can do the inference from Section 2.4.

5.1 Dependency Grammar

Dependency grammar (Tesnire, 1959; Hudson, 1990) is a grammar formalism which is alternative to phrase structure or categorial grammar. Unlike phrase structure grammar. Instead, the nodes in the syntax tree are simply the words of the sentence, and the edges stand for direct *dependency* relations between words.

By way of example, consider Fig. 1.8, the *dependency tree* of the German sentence *Den Mann liebt die Frau* ("the woman loves the man", as in 1.4). At the root of the tree is the finite verb *liebt*. It has two *dependents*, namely the subject *Frau* and the object *Mann*. Each of these has one dependent – the determiners *die* and *den*, respectively. We label the edges with *roles* such as *det*, *subject*, *object*. The roles are taken from a set *Roles*; this set is divided into complement roles *Comps* and modifier roles (e.g. *adj* for adjective modification).

Dependency grammar is particularly interesting for languages with free word order. While phrase structure grammars require special

$$\begin{bmatrix} \textbf{string} : \text{den} \\ \textbf{cats} : \mathit{det} \\ \textbf{agrs} : \langle \mathit{masc}, \mathit{acc} \rangle \\ \textbf{roles} : \emptyset \end{bmatrix} \quad \begin{bmatrix} \textbf{string} : \text{Mann} \\ \textbf{cats} : n \\ \textbf{agrs} : \langle \mathit{masc}, \mathit{nom/acc} \rangle \\ \textbf{roles} : \{\mathit{det}\} \end{bmatrix} \quad \begin{bmatrix} \textbf{string} : \text{Frau} \\ \textbf{cats} : n \\ \textbf{agrs} : \langle \mathit{fem}, \mathit{nom/acc} \rangle \\ \textbf{roles} : \{\mathit{det}\} \end{bmatrix}$$

$$\begin{bmatrix} \textbf{string} : \text{die} \\ \textbf{cats} : \mathit{det} \\ \textbf{agrs} : \langle \mathit{fem}, \mathit{nom/acc} \rangle \\ \textbf{roles} : \emptyset \end{bmatrix} \quad \begin{bmatrix} \textbf{string} : \text{die} \\ \textbf{cats} : \mathit{relpro} \\ \textbf{agrs} : \langle \mathit{fem}, \mathit{nom/acc} \rangle \\ \textbf{roles} : \emptyset \end{bmatrix} \quad \begin{bmatrix} \textbf{string} : \text{liebt} \\ \textbf{cats} : \mathit{vfin} \\ \textbf{agrs} : \langle \_, \mathit{nom} \rangle \\ \textbf{roles} : \{\mathit{subj}, \mathit{np\_acc}\} \end{bmatrix}$$

Figure 1.9: Some lexical entries.

mechanisms for making word order more liberal, word order in dependency grammars is totally free to begin with. Their problem is not to allow more liberal word orders, but to exclude ungrammatical ones. Constraints on word order that tackle some problems that are nontrivial for phrase structure grammars can be integrated into the axiomatization we present below, but we can't go into details here.

5.2 An Example Grammar

A dependency grammar is built from sets of strings, roles, categories, and agreement tuples. It consists of a set of lexicon entries e, for each of which it defines a string $\mathit{string}(e)$, a set of categories $\mathit{cats}(e)$, a set of agreement tuples $\mathit{agrs}(e)$, and a set of roles $\mathit{roles}(e)$ that must be filled by dependents. An example for a tiny lexicon fragment is given in Figure 1.9. We use $\langle \_, \mathit{nom/acc} \rangle$ as shorthand notation for the set of all agreement tuples that have arbitrary gender and nominative or accusative case.

In addition, the grammar defines a *role constraint* Γ_ρ for each role ρ, which must be satisfied for any two nodes in the dependency tree that are linkes by a ρ-edge. For example, objects must be nouns in accusative case that depend on a finite verb:

$$\Gamma_{object}(w, w') \quad =_{def} \quad \mathit{cat}(w) = \mathit{vfin} \wedge \mathit{cat}(w') = n \wedge \mathit{agr}(w') \in \langle \_, \mathit{acc} \rangle$$

As another example, a determiner must agree with the noun it depends on and be the first word in the *yield* of the noun. The yield of a word is the set of all words in the sentence that can be reached by traversing any number of dependency edges.

$$\begin{array}{rcl} \Gamma_{det}(w, w') & =_{def} & \mathit{cat}(w') = \mathit{det} \wedge \mathit{agr}(w) = \mathit{agr}(w') \\ & \wedge & w' = \min(\mathit{yield}(w)). \end{array}$$

Finally, the grammar can define global word order constraints. We ignore these in this presentation.

5.3 Translation to Set Constraints

Now we encode dependency parsing using set constraints. As in Section 4, we encode a (dependency) tree using sets of nodes. We impose a finite set constraint whose models correspond to valid dependency trees of the input sentence. Parsing, then, reduces to finding values for the set variables.

Preliminaries

First of all, we pick unique numbers for all syntactic categories, roles, and agreement tuples. We encode the words in the sentence (i.e. the nodes of the dependency tree) by their linear positions. Let's say that *Nodes* denotes the set of all these positions. In this way, we only need to talk about integers and sets of integers. Each position corresponds to one node in the dependency tree. We associate with every node w the finite domain variables $\mathit{cat}(w)$ (the category of w) and $\mathit{agr}(w)$ (agreement); these variables are constrained to be encodings of categories and agreement tuples, respectively. In addition, we associate the finite set variables $\mathit{roles}(w)$ (role labels of edges starting at w), $\mathit{mothers}(w)$ (set of mothers of w), for each role $\rho \in \mathit{Roles}$, $\rho(w)$ (set of daughters of w via a ρ-labeled edge), and finally $\mathit{daughters}(w)$ (set of daughters of w via any edge), all of which are constrained to be subsets of *Nodes*. We can access the word at position w via the variable $\mathit{String}(w)$.

Lexicon

First, let's connect the nodes to the lexicon. We associate with every word s a sequence $\mathit{Lex}(s) = \langle e_1, \ldots, e_k \rangle$ of possible *lexical entries.* A lexical entry is a record defining the entries *cats* (possible categories), *args* (possible arguments), and *roles*, as above. (Note that nodes have FD-valued features *agr*, *cat*, whereas lexical entries have set-valued features *agrs*, *cats*.) Now for each node w, we introduce two new variables N_w and E_w. N_w will denote the *index* of the lexical entry in $\mathit{Lex}(s)$ that is actually used in w, and E_w is this lexical entry itself (a record). N_w and E_w should be related by a selection constraint. Selection constraints are only defined on sequences of set variables, but we can use the pointwise selection constraints:

$$\begin{array}{rcl} \mathit{cats}(E_w) & = & \langle \mathit{cats}(e_1), \ldots, \mathit{cats}(e_k)\rangle[N_w] \\ \mathit{agrs}(E_w) & = & \langle \mathit{agrs}(e_1), \ldots, \mathit{agrs}(e_k)\rangle[N_w] \\ \mathit{roles}(E_w) & = & \langle \mathit{roles}(e_1), \ldots, \mathit{roles}(e_k)\rangle[N_w] \end{array}$$

Now the connection between nodes and lexical entries can be axiomatized in the following way:

$$\begin{array}{c} \mathit{cat}(w) \in \mathit{cats}(E_w) \\ \mathit{agr}(w) \in \mathit{agrs}(E_w) \\ \mathit{roles}(w) = \mathit{roles}(E_w). \end{array}$$

Well-formedness

Now that we know where each node gets its features from, let's impose constraints on the well-formedness of a dependency tree. First of all, we have to ensure proper *valencies*. For one, we must make sure that whenever a certain role ρ is in the *roles* set of a node w, it is realized by a daughter, and vice versa:

$$\mathsf{equiv}(|\rho(w)|>0,\ \rho \in \mathit{roles}(w)).$$

On the other hand, every complement of a node must be realized by at most one other node. (There is no such restriction for modifiers.) That is, for every $\rho \in \mathit{Comps}$, we have the constraint

$$0 \leq |\rho(w)| \leq 1.$$

To express that the models really encode trees, we say that every node except for the unique root has exactly one mother and that there are no cycles. First, we introduce a single new FD variable ROOT for the identity of the root node of the dependency tree. We define mothers via the $\mathit{mothers}(w)$ and $\mathit{daughters}(w)$ variables introduced above. A daughter is a daughter via an arbitrary edge,

$$\mathit{daughters}(w) = \cup\{\rho(w) \mid \rho \in \mathit{Roles}, w \in \mathit{Nodes}\}$$

and *mothers* is the left inverse relation of *daughters*:

$$\mathsf{equiv}(w \in \mathit{mothers}(w'),\ w' \in \mathit{daughters}(w)).$$

Uniqueness of mothers is axiomatized as follows:

$$\begin{array}{rl} & 0 \leq |\mathit{mothers}(w)| \leq 1 \\ \wedge & \mathsf{equiv}(|\mathit{mothers}(w)|=0,\ w = \textsc{root}). \end{array}$$

Finally, every node is either a daughter or the root:

$$Nodes = \{\textsc{root}\} \uplus \biguplus_{\rho,w} \rho(w).$$

To axiomatize cycle-freeness, we first associate each node w with FS variables $yield(w)$ and $yield!(w)$. The yield of a node w is the set of nodes (including w) which can be reached from w by traversing any number of dependency edges. The set of this nodes is what $yield(w)$ should denote; $yield!(w)$ is the irreflexive counterpart. $yield!$ can be axiomatized by using selection constraints, but we skip this here for lack of space. What we're interested in is that we can impose the FS constraint

$$yield(w) = \{w\} \uplus yield!(w)$$

and obtain both the definition of $yield$ and cycle-freeness (w is not a member of its own irreflexive yield).

Finally, each pair of nodes that is connected by a ρ-labeled edge must satisfy the *role constraint* Γ_ρ, as we said in Section 5.2. We model this by requiring that for all nodes w and w' and for all roles ρ,

$$\mathsf{imply}(w' \in \rho(w),\ \Gamma_\rho(w, w')).$$

Due to the implementation of imply, this constraint acts as a propagator in both directions: If $w' \in \rho(w)$, it will impose the role constraint $\Gamma_\rho(w, w')$; and once $\Gamma_\rho(w, w')$ turns out to be inconsistent, it infers that $w' \notin \rho(w)$.

Putting It Together

We have now axiomatized encodings of trees (this time, dependency trees) using set constraints; these constraints enforce that the trees are well-formed trees for the given sentence and conform to the grammar. As in Section 4, the axioms essentially give us an actual Oz implementation.

This time, distribution will enumerate possible parses of the sentence according to the grammar. Distribution is over the values of the *mothers* variables, the lexicon selection variables N_w, and the role variables $\rho(w)$. That is, we need to distribute both over FD and FS variables.

5.4 Example

To conclude the exposition of the dependency parser, let's discuss the sentence (1.4), repeated below, from Section 2.

(1.9) *Den*	*Mann*	*liebt*	*die*	*Frau.*
the(acc)	man	loves	the(nom/acc)	woman

'The woman loves the man.'

The complete (unique) dependency tree for this sentence is in Fig. 1.8. This tree can be derived purely by propagation. Below, we sketch the propagation steps that the actual implementation would perform. We skip some details for better readability, but all the missing details can in principle be filled in from constraints in this section.

First, since *Frau* requires a *det*-complement, the set $det(Frau)$ is a singleton. It can't contain *Frau*, *Mann*, or *liebt*, because the categories of these words violates the role constraint for *det*. Neither can it be *den*, which doesn't agree with *Frau*. This means that $det(Frau) = \{die\}$. Now the *det* role constraint tells us that the lexical entry we must pick for the *die* is the determiner one. We can derive that the determiner of *Mann* is *den* and that hence, *Mann* has accusative case, in a similar fashion.

Furthermore, the two sets *subject*(*liebt*) and *object*(*liebt*) must be singletons (since *liebt* requires both roles) and can't contain *liebt*, *den*, or *die* (which have the wrong categories). We have just inferred that *Mann* has accusative case, so it can't be a member of *subject*(*liebt*), as this would violate the *subject* role constraint; so the value of *subject*(*liebt*) must be $\{Frau\}$. This disambiguates the case of *Frau* (and of *die*) to nominative, which in turn excludes *Frau* from membership in *object*(*liebt*); so *object*(*liebt*) must be $\{Mann\}$.

An interesting computation based on selection constraints takes place for the (seemingly simpler) sentence (1.10).

(1.10) *Die*	*Frau*	*liebt*	*den*	*Mann.*
the(nom/acc)	woman	loves	the(acc)	man

'The woman loves the man.'

In order to derive that *den* is the determiner for *Mann*, we must exclude *die* from $det(Mann)$. We do this by reasoning about agreement; but the agreement of *die* is defined by the lexical entry, and we don't know at first which lexical entry we're going to use. However, both possible lexical entries contain the same agreement information. So the constructive disjunction provided by the lexical selection constraint will be able to assign this common information to *die*, without even having to decide which lexical entry to use.

5.5 Discussion

We have encoded dependency parse trees using sets and dependency grammars plus general well-formedness conditions as constraints on these sets. The grammar is encoded in the lexicon, role constraints, and in word order constraints that we didn't discuss.

The process of dependency parsing as presented is fundamentally different to traditional chart parsing. A chart parser generates possible (partial) parse trees from smaller ones. Along the way, it may turn out that some of the generated possibilities can't contribute to a parse after all; those must be discarded. The dependency parser presented here doesn't generate trees; it *configures* them from a given set of nodes. It accumulates as much information about the structure of the tree by propagation as possible before distributing choices. (Local) ambiguities are reflected as undetermined variable values and modeled with different types of constraints; e.g., attachment ambiguities with FS constraints (values of the $mothers(w)$ variables), agreement with FD variables, and lexical ambiguities with selection constraints.

In practice, it turns out that the parser is very efficient when using a small but interesting grammar of German; again, it produces search trees with few if any failures. However, syntactic coverage is far away from what is available for more widely used grammar formalisms (say, LFG, HPSG, TAG). The development of a larger-scale grammar and exploration of parsing times on this grammar are important lines of future work. Another interesting problem is to see to what extent ideas from this implementation carry over to other grammar formalisms. A start in this direction is (Duchier and Thater, 1999), which presents a parser for D-Tree Grammars (Rambow et al., 1995), a variant of TAG.

6 Conclusion

In this paper, we have discussed the basic ideas of constraint programming and argued that it can be useful for applications in computational linguistics. Constraint programming is an approach to combinatorial problems that attempts to reduce the size of the search space by constraint propagation. Put another way, it prefers model elimination over model enumeration. So far, the main applications of constraint programming in computational linguistics concern the treatment of ambiguities; local ambiguities, which occur only in intermediate processing steps, can often be cheaply disambiguated by propagation alone.

The idea of propagation is extremely natural. In fact, there are probably very few pure generate-and-test programs that do not "propagate" even a little between case distinctions. What constraint programming can offer here is a more general framework that allows a clean specification of propagators. Some of the concrete implementation ideas in this paper have been developed independently elsewhere as well. For instance, Maxwell and Kaplan (1991) propose a treatment of disjunction that employs something very much like choice variables.

We have presented two examples from computational linguistics to which constraint programming has been applied successfully: scope underspecification with tree descriptions and parsing for dependency grammars. Our intention is to encourage computational linguists to look into constraint programming for their own needs. We are only beginning to understand what characterizes a problem where constraint programming can be more useful than traditional methods (say, charts); but the examples suggest that it can indeed allow elegant, efficient solutions to old problems.

A situation that constraint programming cannot deal with very well (yet) is when new objects must be created dynamically. Many existing data structures and algorithms in computational linguistics rely on this; examples are phrase structure parsers, which create nonterminal nodes as they go along, and all forms of generation algorithms. Another interesting question is if, and how, constraint programming can be brought together with statistical methods. This is as unclear for constraint programming as for any other logic-based method. Both questions are open problems.

References

Aggoun, A. and N. Beldiceanu. 1993. Overview of the CHIP compiler system. In Benhamou, F. and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 421–437. The MIT Press, Cambridge, MA.

Aggoun, A., D. Chan, P. Dufresne, E. Falvey, H. Grant, A. Herold, G. Macartney, M. Meier, D. Miller, S. Mudambi, B. Perez, E. Van Rossum, J. Schimpf, P. A. Tsahageas, and D. H. de Villeneuve. 1995. ECL^{i}PS^e 3.5. User manual, European Computer Industry Research Centre (ECRC), Munich, Germany.

Apt, K. 1990. Logic programming. In Leeuwen, J. v., editor, *Handbook of Theoretical Computer Science*, volume B, pages 493–571. The MIT Press.

Backofen, R., J. Rogers, and K. Vijay-Shanker. 1995. A first-order axiomatization of the theory of finite trees. *Journal of Logic, Language, and Information*, 4:5–39.

Bos, J. 1996. Predicate logic unplugged. In *Proceedings of the 10th Amsterdam Colloquium*, pages 133–143.

Caseau, Y. and F. Laburthe. 1996. Introduction to the CLAIRE programming language. Technical report, Departement Mathmatiques et Informatique, Ecole Normale Suprieure, Paris, France. Version 1.0.

Cornell, T. 1994. On determining the consistency of partial descriptions of trees. In *Proceedings of ACL*.

Dincbas, M., P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. 1988. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems FGCS-88*, pages 693–702, Tokyo, Japan.

Duchier, D. 1999. Axiomatizing dependency parsing using set constraints. In *Sixth Meeting on Mathematics of Language*, Orlando, Florida.

Duchier, D. and C. Gardent. 1999. A constraint-based treatment of descriptions. In *Proceedings of IWCS-3*, Tilburg.

Duchier, D., C. Gardent, and J. Niehren. 1999. Concurrent constraint programming in Oz for natural language processing. Lecture notes, `http://www.ps.uni-sb.de/~niehren/oz-natural-language-script.html`.

Duchier, D. and J. Niehren. 2000. Dominance constraints with set operators. In *Proceedings of the First International Conference on Computational Logic (CL2000)*, LNCS. Springer.

Duchier, D. and S. Thater. 1999. 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.

Egg, M., J. Niehren, P. Ruhrberg, and F. Xu. 1998. Constraints over Lambda-Structures in Semantic Underspecification. In *Proceedings COLING/ACL'98*, Montreal.

Egg, M., A. Koller, and J. Niehren. 2000. The constraint language for lambda structures. Journal Submission.

Gardent, C. and B. Webber. 1998. Describing discourse semantics. In *Proceedings of the 4th TAG+ Workshop*, Philadelphia. University of Pennsylvania.

Groenendijk, J. and M. Stokhof. 1991. Dynamic predicate logic. *Linguistics & Philosophy*, 14:39–100.

Hudson, R. 1990. *English Word Grammar*. Basil Blackwell, Oxford.

ILOG. 1999. *ILOG Solver: User's Manual.* Gentilly, France. Version 4.4.

Jaffar, J. and J.-L. Lassez. 1987. Constraint logic programming. In *ACM Symposium on Principles of Programming Languages*, pages 111–119. The ACM Press.

Janson, S. and S. Haridi. 1991. Programming paradigms of the Andorra Kernel Language. In Saraswat, V. and K. Ueda, editors, *Proceedings of the 1991 International Symposium on Logic Programming*, pages 167–186, San Diego, California.

Kamp, H. and U. Reyle. 1993. *From Discourse to Logic.* Kluwer, Dordrecht.

Koller, A., K. Mehlhorn, and J. Niehren. 2000. A polynomial-time fragment of dominance constraints. Submitted. `http://www.coli.uni-sb.de/~koller/papers/poly-dom.html`.

Koller, A. and J. Niehren. 1999. Scope underspecification and processing. Lecture Notes, ESSLLI 99, Utrecht. `http://www.coli.uni-sb.de/~koller/papers/esslli99.html`.

Koller, A. and J. Niehren. 2000. On underspecified processing of dynamic semantics. In *Proceedings of COLING-2000*, Saarbrücken.

Koller, A., J. Niehren, and R. Treinen. 1998. Dominance constraints: Algorithms and complexity. In *Proceedings of the Third Conference on Logical Aspects of Computational Linguistics*, Grenoble.

Marcus, M. P., D. Hindle, and M. M. Fleck. 1983. D-theory: Talking about talking about trees. In *Proceedings of the 21st ACL*, pages 129–136.

Marriott, K. and P. J. Stuckey. 1998. *Programming with Constraints.* Kluwer Academic Publisher.

Maxwell, J. T. and R. Kaplan. 1991. A method for disjunctive constraint satisfaction. In Tomita, M., editor, *Current Issues in Parsing Technology*, pages 173–190. Kluwer.

Mozart Consortium. 1999. The Mozart Programming System web pages. `http://www.mozart-oz.org/`.

Muskens, R. 1995. Order-Independence and Underspecification. In Groenendijk, J., editor, *Ellipsis, Underspecification, Events and More in Dynamic Semantics.* DYANA Deliverable R.2.2.C.

Rambow, O., K. Vijay-Shanker, and D. Weir. 1995. D-Tree Grammars. In *Proceedings of ACL'95.*

Reyle, U. 1993. Dealing with ambiguities by underspecification: construction, representation, and deduction. *Journal of Semantics*, 10:123–179.

Saraswat, V. A. and M. Rinard. 1990. Concurrent constraint programming. In *ACM Symposium on Principles of Programming Languages*, pages 232–245. The ACM Press.

Saraswat, V. A., M. Rinard, and P. Panangaden. 1991. Semantic foundations of concurrent constraint programming. In *ACM Symposium on Principles of Programming Languages*, pages 333–352. ACM Press, New York.

Schulte, C. 2000. Programming deep concurrent constraint combinators. In Pontelli, E. and V. S. Costa, editors, *Practical Aspects of Declarative Languages*, volume 1753 of *LNCS*, pages 215–229. Springer-Verlag.

Smolka, G. 1995. The Oz programming model. In van Leeuwen, J., editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin.

Smolka, G. 1998. Concurrent constraint programming based on functional programming. In Hankin, C., editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, vol. 1381, pages 1–11, Lisbon, Portugal. Springer-Verlag.

Tesnire, L. 1959. *Elments de syntaxe structurale*. Klincksiek, Paris.

van Deemter, K. and S. Peters, editors. 1996. *Semantic Ambiguity and Underspecification*. CSLI Publications, Stanford.

Vijay-Shanker, K. 1992. Using descriptions of trees in a tree adjoining grammar. *Computational Linguistics*, 18:481–518.