Towards a Constraint Parser for Categorial Type Logics

Marco Kuhlmann

Master of Science
School of Cognitive Science
Division of Informatics
University of Edinburgh

2002

Abstract

This thesis shows how constraint programming can be applied to the processing of Categorial Type Logics (CTL). It presents a novel formalisation of the parsing task for categorial grammars as a tree configuration problem, and demonstrates how a recent proposal for *structural constraints* on CTL parse trees [11] can be integrated into this framework. The resulting processing model has been implemented using the Mozart/Oz programming environment. It appears to be a promising starting point for further research on the application of constraint parsing to CTL and the investigation of the practical processing complexity of CTL grammar fragments.

Acknowledgements

I would like to thank Mark Steedman for supervising this project, and Katrin Erk and Geert-Jan Kruijff for getting me interested in Categorial Type Logics in the first place.

My work on this thesis has greatly profited from discussions with Jason Baldridge, Ralph Debusmann, Denys Duchier, Katrin Erk, Geert-Jan Kruijff, Joachim Niehren, and Guido Tack. Katrin, Geert-Jan and Guido read drafts of individual chapters of this thesis and provided me with helpful comments. Guido also modified his TkTreeWidget to suit my needs, and was a very pleasant and tolerant host during three weeks in Saarbrücken.

Many thanks go to my coursemates at 2 Buccleuch Place, for all those days and nights in the Drawing Room and elsewhere: Colin Bannard, Chris Callison-Burch, Nicola Cathcart, Dave Hawkey, Danielle Matthews, Matthijs van der Meer, Sebastian Pádo, Daniel Roberts, and David Rojas.

I gratefully acknowledge the two scholarships I received during my time in Scotland, from the Deutscher Akademischer Austauschdienst and the Studienstiftung des Deutschen Volkes.

My family was a constant source of support both during this last year in Edinburgh and the three previous years in Saarbrücken. Without them, all this would have hardly been possible.

My deepest gratitude I owe to Antje, who has been so patient with me.

Edinburgh, September 2002

Marco Kuhlmann

Declaration

I declare that this thesis has been composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Marco Kuhlmann

Contents

Chapter 1

Introduction

Categorial Type Logics [25] are an appealing framework for the formal description of human language grammar. Unfortunately, up to the present day, no efficient parsers exist for them. This thesis suggests constraint programming [23, 31] as a promising technique to change this situation. More specifically, it provides answers to the following two questions:

Question 1. How can the problem of parsing Categorial Type Logics be formulated such that constraint programming can be applied to it?

Question 2. Given such a formulation, can we formalise and implement constraints that efficiently prune the search space of the problem?

The first question asks for a formulation of the parsing problem of Categorial Type Logics as a constraint satisfaction problem. Previous work has focused on chart-based approaches [15, 18] or advanced theorem proving techniques such as proof nets [27]. Both approaches have their shortcomings: Chart-based techniques do not scale up to the general, multimodal form of Categorial Type Logics; proof net algorithms have exponential complexity and are therefore useful only for the development of small grammars. This thesis presents a novel and efficient encoding of the parsing problem of CTL grammars into the paradigm of constraint programming, drawing from current research on constraint parsing with tree descriptions [10] and its application to declarative flavours of dependency grammar [8, 7]. The encoding opens up a whole range of new techniques and tools to the developer of realistic parsers for Categorial Type Logics.

These techniques and tools are used to positively answer the second question: This thesis shows that a recent proposal for constraints on Categorial Type Logics [11] based on the methodology of abstract interpretation [6] can be formalised within the proposed framework. The resulting implementation, which is integrated into the Mozart/Oz development environment, can serve as a "workbench" for the study of the usage of constraints for categorial grammars, and a first step towards fully-fledged parsers. The only tool currently available for this task is Richard Moot's Grail system [26], which bases on proof-net algorithms that cannot avoid the exponential complexity that comes along with the parsing problem of arbitrary CTL fragments.

The remainder of this thesis is organised as follows:

Chapter 2 will introduce categorial type logics and review previous approaches to implementing them. The focus of the exposition will be on the issue of *structural control*, the management of linguistic resources.

Chapter 3 introduces constraint programming. It shows the close links between the central implementation technique of constraint propagation and the formal work on abstract interpretation. It also reviews the high-level abstractions for constraint programming provided by the Mozart/Oz system.

Chapter 4 shows how parsing categorial grammars can be formalised as a constraint satisfaction problem, and demonstrates the integration of structural constraints into this formalisation by an example. The implementation of this formal framework is discussed in chapter 5.

Finally, chapter 6 concludes with a brief discussion of what has been achieved by this project and which questions have been left unanswered.

Chapter 2

Categorial Type Logics

This chapter introduces Categorial Type Logics (CTL) [25], a powerful framework for the description of natural language grammar. From other categorial formalisms, CTL are distinguished by their deductive view on grammatical composition, based on a fully developed model and proof theory. Modern flavours of CTL offer a fine-grained control over the permitted syntactic structures. The challenge that comes along with the descriptive power of CTL is their potential computational complexity.

The plan of this chapter is as follows: In the first section, an informal overview of the basics of categorial grammars is given; for a more comprehensive presentation, the reader is referred to the introductory survey of Wood [36]. Section 2.2 introduces the Lambek calculi, a family of logical systems that stake off the logical landscape of Categorial Type Logics in terms of their resource sensitivity. An example for a multimodal CTL framework [14] is given in section 2.3. Finally, section 2.4 presents previous work on parsing CTL and discusses the complexity issues related to it.

2.1 Categories and combinations

The foundation of Categorial Grammar (CG) was laid in the seminal of Ajdukiewicz [1], which draws from earlier work by Lesniewski [22], Frege, Husserl, and Russell. Ajdukiewicz's original goal was the formalisation of what he called the "syntactic connexion" – the well-formedness of syntactic structures.

Categories

At the heart of CG there is a distinction between *atomic* and *complex* types of linguistic signs: Atomic categories are "complete", complex categories "incomplete" types in the sense that they can combine with other signs. This distinction is similar to the one between atomic and functional types present in type theory, and by means of the Curry–Howard correspondence it is indeed straightforward to assign semantic types to syntactic categories (see section 2.4 for examples).

The set of assumed atomic categories differs from one author to the other, but usually at least s (for sentences) and np (for noun phrases) are counted hereunder. In type theory, these correspond to the type of truth values t and the entity type e.

Complex categories are constructed inductively by means of a small set of type-forming operators. Ajdukiewicz used a single, fraction-style operator; two categories $\frac{A}{B}$ and B could combine by "multiplication" to form the new category A. Lambek [21] established a directed notation, distinguishing two "slash operators" $\backslash$ and $/$ that differ in terms of the direction in which the combination takes place. This gives rise to the following two combination rules:[1]

$$B, B\backslash A \Rightarrow A \qquad A/B, B \Rightarrow A$$

Typical examples of complex categories are verbs, which can be regarded as being "incomplete" in that they require a number of noun phrases to become a complete sentence. For instance, transitive verbs are usually assigned the category $(np\backslash s)/np$ or $np\backslash(s/np)$.

[1] Notice that here, as in the rest of this thesis, the original "result on top" notation of Lambek for categories is used, rather than the "result leftmost" notation of e. g. Steedman [34].

$$\dfrac{\dfrac{\text{Dan}}{np} \quad \dfrac{\dfrac{\text{likes}}{(np\backslash s)/np} \quad \dfrac{\text{parsnips}}{np}}{np\backslash s}}{s}$$

Figure 2.1: AB grammar derivation of "Dan likes parsnips"

AB grammar

Given the two combination schemata from above and appropriate lexical assignments, a number of simple English sentences can be derived (see figure 2.1 for an example). Turning a CG derivation upside down, one immediately recognises the similarities to a parse tree in a context-free phrase structure grammar. In terms of generative capacity, this most basic form of categorial grammar, which after Ajdukiewicz and Bar-Hillel is called *AB grammar*, is indeed equivalent to context-free grammars. Notice however, that CG is highly lexicalised and comes with only two rule schemata, whereas a phrase structure grammar normally involves a much larger inventory of rules.

It is well known that context-free grammars – and therefore, by equivalence, AB grammar – are not powerful enough to describe many linguistic phenomena. Within the research on categorial grammars, there have been two strands of approaches to remedy this deficiency, referred to as the *rule-based* and the *deductive* style of CG. The remainder of this section briefly discusses a rule-based extension of AB grammar. The deductive approach will be discussed in detail throughout the rest of this chapter.

Structural control

In rule-based extensions of AB grammar, the basic combination schemata of CG are amended by other type-changing rules. To give a concrete example, table 2.1 shows an (incomplete) set of the rules of Combinatory Categorial Grammar CCG [34].

$X/Y, Y \Rightarrow X$	forward application (>)
$Y, Y\backslash X \Rightarrow X$	backward application (<)
$X, conj, X \Rightarrow X$	conjunction (Conj)
$X/Y, Y/Z \Rightarrow X/Z$	forward composition (>B)
$Z\backslash Y, Y\backslash X \Rightarrow Z\backslash X$	backward composition (<B)
$X \Rightarrow Y/(X\backslash Y)$	forward type-raising (>T)
$X \Rightarrow (Y/X)\backslash Y$	backward type-raising (<T)
$(X/Y)/Z, Y/Z \Rightarrow X/Z$	forward substitution (>S)
$Z\backslash Y, Z\backslash(Y\backslash X) \Rightarrow Z\backslash X$	backward substitution (<S)

Table 2.1: (Some of) the rules of CCG [34]

The selection of rules is a delicate matter: On one side, they must be powerful enough to allow the derivation of the sentence structures one wants to permit; this will increase the generative capacity of the grammar. On the other side, the rules must not be too expressive, so that they do not allow more structural analyses than wanted, and can be processed efficiently. This tension, which will be the central theme of this chapter, may be referred to as the issue of *structural control*.

At a first glance, the structural control regime of CCG seems to be to permissive: Figure 2.2 shows an example of what the literature has called "spurious ambiguities" – two analyses of the same sentence that (employing

$$\dfrac{\text{Dan}_{np} \qquad \dfrac{\text{likes}_{(np\backslash s)/np} \qquad \text{parsnips}_{np}}{np\backslash s}\,(>)}{s}\,(<)$$

(a) Dan (likes parsnips)

$$\dfrac{\dfrac{\text{Dan}_{np}}{s/(np\backslash s)}\,(>\text{T}) \qquad \text{likes}_{(np\backslash s)/np}}{s/np}\,(>\text{B}) \qquad \dfrac{\text{parsnips}_{np}}{s}\,(>)$$

(b) (Dan likes) parsnips

Figure 2.2: "Spurious ambiguities" in CCG

the Curry–Howard correspondence) share the same semantics. Steedman argues, however, that the two analyses indeed *are* different, when considered on the level of information structure [34]. Take the following two sentences:

(1) *Dan* likes parsnips; *I* detest them.

(2) Dan likes *parsnips*, not *swedes*.

The first halves of both sentences are syntactically and semantically the same, yet in a broader sense, they have different meanings: In (1), *Dan* carries a focus; in (2), the focus is on the *parsnips*. Steedman's framework for the treatment of differences like this will derive (1) as in figure 2.2(a), and (2) as in figure 2.2(b).

Thus, with regards to information structure, the "spurious ambiguities" are no ambiguities at all. Nevertheless, for many applications, we might be interested only in a traditional semantic analysis of a sentence. In this case, it would be desirable to adjust the "granularity" of the structural control and consider for example derivations modulo equal semantics (see section 2.4). For rule-based approaches like CCG, this can only be done on the implementation level. The next section sets the stage for formalisms that allow a change of granularity *within* the grammar itself.

2.2 The Lambek calculi

The basic rules of category combination, which in CCG are regarded as functional applications, can also be seen as directed versions of the *modus ponens*:

$$\frac{B \quad B\backslash A}{A} \qquad \frac{A/B \quad B}{A}$$

To ease the reasoning about these elementary rules, the "slashed" premises of a *modus ponens* will be called *rides*, and the "non-slashed" premises *tickets*. The intuition behind this terminology is that if one has a ticket B, then one can make the ride $B\backslash A$ to arrive at A.

Grammars as logics

Taking the *modus ponens* view on the basic combination rules, a series of other rules naturally suggests itself in analogy to other logical calculi: The rules that eliminate the slash operators can be complemented by rules that introduce them. Additionally, the juxtaposition of ticket and ride can be made explicit by introducing a binary concatenation or *product* operator $\bullet$. This implies a refinement of the notion of a category, which is formalised in definition 2.1 below.

Definition 2.1 (Lambek-style categories) Given a set of atomic types $\mathcal{A}$, the set $\mathcal{C}_\mathcal{A}$ of *categories over* $\mathcal{A}$ is defined by the following abstract syntax:

$$\mathcal{C}_\mathcal{A} ::= \mathcal{A} \mid \mathcal{C}_\mathcal{A} \bullet \mathcal{C}_\mathcal{A} \mid \mathcal{C}_\mathcal{A} \backslash \mathcal{C}_\mathcal{A} \mid \mathcal{C}_\mathcal{A} / \mathcal{C}_\mathcal{A}.$$

When $\mathcal{A}$ is arbitrary or obvious from the context, it is usually omitted.

The logical aspects of categorial grammar were first developed in Lambek's work in the late 1950's [21], which became the foundation of the *deductive* style of categorial grammar, which emphasises the connections between grammatical analysis and proof theory: The grammaticality of a sentence can be proved by means of the inference rules of a *categorial logic*. Different categorial logics are distinguished in terms of their structural control.

The rules of the most restrictive of these categorial logics, the *non-associative Lambek calculus* NL, is given in figure 2.3. The presentation is in Natural Deduction format and uses sequents (Γ, C) according to definition 2.2.

Definition 2.2 (Sequent) A *sequent* is a pair (Γ, C) of a binary tree over categories Γ, called the *antecedent*, and a single category C, called the *succedent*.

In figure 2.3, sequents are written $\Gamma \vdash C$ and can be read as "the grammatical structure Γ licenses the category C". The notation $\Gamma[\Delta]$ in the conclusion

$$\frac{}{A \vdash A}\ (\text{Ax})$$

$$(/\text{I})\ \frac{(\Gamma, B) \vdash A}{\Gamma \vdash A/B} \qquad \frac{\Gamma \vdash A/B \quad \Delta \vdash B}{(\Gamma, \Delta) \vdash A}\ (/\text{E})$$

$$(\backslash\text{I})\ \frac{(B, \Gamma) \vdash A}{\Gamma \vdash B\backslash A} \qquad \frac{\Gamma \vdash B \quad \Delta \vdash B\backslash A}{(\Gamma, \Delta) \vdash A}\ (\backslash\text{E})$$

$$(\bullet\text{I})\ \frac{\Gamma \vdash A \quad \Delta \vdash B}{(\Gamma, \Delta) \vdash A \bullet B} \qquad \frac{\Delta \vdash A \bullet B \quad \Gamma[(A, B)] \vdash C}{\Gamma[\Delta] \vdash C}\ (\bullet\text{E})$$

Figure 2.3: Non-associative Lambek calculus NL [25, p. 110 f.]

of the product elimination rule (•E) refers to a distinguished subtree Δ in Γ, which, reading the rule from its bottom to the top, in the premise gets replaced by the subtree (A, B).

Dimensions of structural control

Pentus proved that NL is context-free [29]. It is therefore interesting to notice, that the CCG rule of type-raising can be derived as a *lemma* in NL (figure 2.4(a)), a "secondary rule" that does not need to be stipulated. This is not possible in AB grammar, although AB grammar is context-free as well. If we would be able to also proof functional composition, then the "spurious ambiguities" of CCG could be replicated in NL. This proof fails, however:

$$\frac{\dfrac{\text{FAILED}}{((A/B, B/C), C) \vdash A}}{(A/B, B/C) \vdash A/C}\ (/\text{I})$$

The failure in the proof attempt sheds some more light on structural control in NL. After the (/I) rule, one would want to apply the (/E) rule, using the C ticket for the B/C ride to obtain B, which could then be used as a ticket for the A/B ride. But B/C and C do not lie in adjacent subtrees of the antecedent structure. In order for the (/E) rule to be applicable, one would first have to *reorder* the antecedent.

$$\dfrac{\dfrac{A \vdash A \quad A\backslash B \vdash A\backslash B}{(A, A\backslash B) \vdash B}\,(\backslash\text{E})}{A \vdash B/(A\backslash B)}\,(/\text{I})$$

(a) Forward type-raising

$$\dfrac{\dfrac{A/B \vdash A/B \qquad \dfrac{B/C \vdash B/C \quad C \vdash C}{(B/C, C) \vdash B}\,(/\text{E})}{(A/B, (B/C, C)) \vdash A}\,(/\text{E})}{\dfrac{((A/B, B/C), C) \vdash A}{(A/B, B/C) \vdash A/C}\,(/\text{I})}\,(\text{Ass2})$$

(b) Forward functional composition

Figure 2.4: Lemmata in NL

Inference rules that accomplish such a reordering are the rules of *associativity* given in figure 2.5(a). As these rules operate only on the antecedent structures, they are referred to as *structural rules*. Adding them to NL yields the *associative Lambek calculus* L, which corresponds to the system originally presented by Lambek [21]. In L, functional composition is provable as a theorem (figure 2.4(b)). This shows that an associative structural regime is crucial for the question of whether or not a logic will allow the derivation of "spurious ambiguities" à la CCG.

A second relevant structural rule is *permutation* (figure 2.5(b)). It is needed for example to explain scrambling phenomena in languages like German, in which verb complements (within certain limits) can permute freely.

The four Lambek calculi resulting from the different combinations of the associativity and permutation rules are shown in figure 2.6. These calculi stake off the borders of the landscape of formalisms considered to be linguistically relevant categorial logics. However, none of them is a realistic formalism for the description of natural language. The *Lambek–van Benthem*

$$\dfrac{\Gamma[((\Delta_1, \Delta_2), \Delta_3)] \vdash A}{\Gamma[(\Delta_1, (\Delta_2, \Delta_3))] \vdash A}\,(\text{Ass1}) \qquad \dfrac{\Gamma[(\Delta_1, (\Delta_2, \Delta_3))] \vdash A}{\Gamma[((\Delta_1, \Delta_2), \Delta_3)] \vdash A}\,(\text{Ass2})$$

(a) associativity

$$\dfrac{\Gamma[(\Delta_2, \Delta_1)] \vdash A}{\Gamma[(\Delta_1, \Delta_2)] \vdash A}\,(\text{Per})$$

(b) permutation

Figure 2.5: Structural rules for permutation and associativity

	−(Per)	+(Per)
−(Ass)	NL	NLP
+(Ass)	L	LP

Figure 2.6: The Lambek calculi

calculus LP for example, which includes both associativity and permutation, is far too permissive for linguistic applications: With every analysis of a sentence, it also licenses all its permutations.

2.3 Multimodal systems

Multimodal categorial logics make use of a refined notion of grammatical structure to overcome the deficiency of the classical Lambek calculi. Categories are now labelled by *structural modes* μ that correspond to different *structural layers* or "granularities" of linguistic description: Structural rules applicable to one mode need not necessarily be applicable to another. In the sequent presentation of multimodal systems, an antecedent has the structure $(X, Y)^\mu$, indicating that the substructures X and Y have been combined in mode μ.

A hybrid Lambek calculus

One example for the multimodal approach is the hybrid system of Hepple [14] given in figure 2.7. It consists of four different structural layers, corresponding to the four different Lambek calculi NL, NLP, L, and LP. Each of these modes μ has its own slash operators $\backslash_\mu$ and $/_\mu$ and its own concatenation operator $\bullet_\mu$. The inference rules of the system, which are the same as in NL, apply globally. The structural rules do not: Permutation is only available in the NLP and LP modes, associativity in the L and LP modes. A

mode-changing rule controls the interaction between different structural layers: Structures with a mode μ may be treated as structures with mode ν if ν is accessible from μ through the accessibility relation $\prec$. Hepple defines $\prec$ as a partial order $\{\mathsf{NL}\} \prec \{\mathsf{NLP}, \mathsf{L}\} \prec \{\mathsf{LP}\}$, based on the intuition that if a structure has been derived with strong structural restrictions, it can surely be derived in a laxer regime. One can also assumes the exactly opposite ordering of modes, allowing less restricted structures to be moved to stricter modes. Interestingly, both assumptions can be justified [25, p. 130].

To illustrate some of the benefits of the multimodal framework, consider the following example sentence from Hepple [14], in which two "gapping" constituents are coordinated before they find their (common) "filler":

(3) Mary spoke and Susan whispered to Bill.

A partial derivation of this sentence is given in figure 2.8. It shows that *Mary spoke* and (as the lexical categories are the same) *Susan whispered* can both be derived as $s/_{\mathsf{L}}pp$. If we assume *and* to have $(s/_{\mathsf{L}}pp)\backslash((s/_{\mathsf{L}}pp)/(s/_{\mathsf{L}}pp))$ among its lexical categories, then this is just what they need to enter coordination and finally combine with the *pp to Bill* to yield a grammatical sentence. Notice that this would not have been possible if *spoke* could have only been used in its original, non-associative mode NL, as the (Ass2) is not available there. On the other hand, due to the mode-changing rule, there is no need for *spoke* to have different lexical entries for each of the modes it is supposed to be used in. The multimodal framework thus allows for economy in the lexicon without giving up structural flexibility.

Extensions

Moortgat [25] extends the multimodal framework by introducing *unary operators* $\Diamond$ and $\Box^{\downarrow}$ that act as boundaries for "domains of locality". A structure can be "locked" and "unlocked" by rules introducing or eliminating unary

$$\frac{}{A \Rightarrow A}\ (\text{Ax})$$

$$\frac{(\Gamma, B)^{\mu} \vdash A}{\Gamma \vdash A/^{\mu}B}\ (/\text{I}) \qquad \frac{\Gamma \vdash A/^{\mu}B \quad \Delta \vdash B}{(\Gamma, \Delta)^{\mu} \vdash A}\ (/\text{E})$$

$$\frac{(B, \Gamma)^{\mu} \vdash A}{\Gamma \vdash B\backslash^{\mu}A}\ (\backslash\text{I}) \qquad \frac{\Gamma \vdash B \quad \Delta \vdash B\backslash^{\mu}A}{(\Gamma, \Delta)^{\mu} \vdash A}\ (\backslash\text{E})$$

$$\frac{\Gamma \vdash A \quad \Delta \vdash B}{(\Gamma, \Delta)^{\mu} \vdash A \bullet_{\mu} B}\ (\bullet\text{I}) \qquad \frac{\Delta \vdash A \bullet_{\mu} B \quad \Gamma[(A, B)^{\mu}] \vdash C}{\Gamma[\Delta] \vdash C}\ (\bullet\text{E})$$

$$\frac{\Gamma[(\Delta_2, \Delta_1)^{\mu}] \vdash A}{\Gamma[(\Delta_1, \Delta_2)^{\mu}] \vdash A}\ (\textsc{Per}),\ \mu \in \{\mathsf{NLP}, \mathsf{LP}\}$$

$$\frac{\Gamma[((\Delta_1, \Delta_2)^{\mu}, \Delta_3)^{\mu}] \vdash A}{\Gamma[(\Delta_1, (\Delta_2, \Delta_3)^{\mu})^{\mu}] \vdash A}\ (\text{Ass1}),\ \mu \in \{\mathsf{L}, \mathsf{LP}\}$$

$$\frac{\Gamma[(\Delta_1, (\Delta_2, \Delta_3)^{\mu})^{\mu}] \vdash A}{\Gamma[((\Delta_1, \Delta_2)^{\mu}, \Delta_3)^{\mu}] \vdash A}\ (\text{Ass2}),\ \mu \in \{\mathsf{L}, \mathsf{LP}\}$$

$$\frac{\Gamma[\Delta^{\mu}]}{\Gamma[\Delta^{\nu}]}\ (\prec),\ \mu \prec \nu$$

Figure 2.7: Multimodal CTL [cf. 14]

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{Mary}}{np \vdash np} \qquad \dfrac{\overline{\text{spoke}}}{(np\backslash_{\mathsf{NL}}s)/_{\mathsf{NL}}pp \vdash (np\backslash_{\mathsf{NL}}s)/_{\mathsf{NL}}pp} \qquad \dfrac{}{pp \vdash pp}}{((np\backslash_{\mathsf{NL}}s)/_{\mathsf{NL}}pp, pp)^{\mathsf{NL}} \vdash np\backslash_{\mathsf{NL}}s}\ (/\text{E})}{(np, ((np\backslash_{\mathsf{NL}}s)/_{\mathsf{NL}}pp, pp)^{\mathsf{NL}})^{\mathsf{NL}} \vdash s}\ (\backslash\text{E})}{(np, ((np\backslash_{\mathsf{NL}}s)/_{\mathsf{NL}}pp, pp)^{\mathsf{L}})^{\mathsf{NL}} \vdash s}\ (\prec)}{(np, ((np\backslash_{\mathsf{NL}}s)/_{\mathsf{NL}}pp, pp)^{\mathsf{L}})^{\mathsf{L}} \vdash s}\ (\prec)}{((np, (np\backslash_{\mathsf{NL}}s)/_{\mathsf{NL}}pp)^{\mathsf{L}}, pp)^{\mathsf{L}} \vdash s}\ (\text{Ass2})}{(np, (np\backslash_{\mathsf{NL}}s)/_{\mathsf{NL}}pp)^{\mathsf{L}} \vdash s/^{\mathsf{L}}pp}\ (/\text{I})$$

Figure 2.8: Non-constituent coordination [cf. 14]

operators, and within locked structures, structural control may be more or less constrained than in the unlocked case. Like their binary counterparts, unary operators can appear in several modes, enabling a very fine-grained structural control.

Another extension to multimodal CTL is to introduce a notion of *dependency* [25, p. 129]. In the multimodal setting, each operator $\circ_\mu$ would occur in two variants, $\circ_\mu^\triangleleft$ and $\circ_\mu^\triangleright$, depending on whether the left or the right hand side of the operator provides the *head* of the resulting structure in the sense of dependency grammar. This additional dimension makes it possible to more adequately describe situations in which a structure semantically acts as a functor, while syntactically it would be regarded as a dependent.

2.4 Efficient structural control

This last section will present some previous approaches to parsing categorial type logics. The deductive view on categorial grammar immediately suggests the application of techniques known from automated theorem proving to this problem. However, as many CTL formalisms are extremely expressive, efficient implementations cannot always be obtained.

Gentzen presentation and Cut elimination

In the context of CTL, the parsing problem can be formulated as a problem of *proof search*: To parse a sentence consisting of words $w_1, \ldots, w_n$, find the admissible derivations of the sequent $c_1, \ldots, c_n \vdash s$, where c_i refers to the category of the word w_i as specified in the lexicon.

The Natural Deduction presentation of the Lambek calculi that was given in figure 2.3 is not suitable for proof search. The problem lies in the elimination rules. For example, to prove the sequent $(\Gamma, \Delta) \vdash A$, one may pick *any* "link formula" B and prove the premises $\Gamma \vdash A/B$ and $\Delta \vdash B$. As there are infinitely many such formulae, proof search will never terminate.

$$\text{(Ax)}\ \frac{}{A \Rightarrow A} \qquad \frac{\Delta \Rightarrow A \quad \Gamma[A] \Rightarrow C}{\Gamma[\Delta] \Rightarrow C}\ \text{(Cut)}$$

$$\text{(/R)}\ \frac{(\Gamma, B) \Rightarrow A}{\Gamma \Rightarrow A/B} \qquad \frac{\Gamma[A] \Rightarrow C \quad \Delta \Rightarrow B}{\Gamma([A/B, \Delta]) \Rightarrow C}\ \text{(/L)}$$

$$\text{(\R)}\ \frac{(B, \Gamma) \Rightarrow A}{\Gamma \Rightarrow B\backslash A} \qquad \frac{\Gamma[A] \Rightarrow C \quad \Delta \Rightarrow B\backslash A}{\Gamma[(\Delta, B\backslash A)] \Rightarrow C}\ \text{(\L)}$$

$$\text{(•R)}\ \frac{\Gamma \Rightarrow A \quad \Delta \Rightarrow B}{(\Gamma, \Delta) \Rightarrow A \bullet B} \qquad \frac{\Gamma[(A, B)] \Rightarrow C}{\Gamma[A \bullet B] \Rightarrow C}\ \text{(•L)}$$

Figure 2.9: NL: Gentzen presentation [25, p. 106]

It is a well-known result from other logical calculi that such indeterminism can be tackled by a *Gentzen presentation* of the calculus, in which the transitivity implicit in the elimination rules is localised into a new rule (Cut), and the search space for link formulae is restricted to subformulae of the antecedent. One then tries to show that each proof involving a (Cut) rule can be reduced to a proof of the same sequent with all the (Cut) rules eliminated. This is called the *Cut elimination* property. In the absence of any structural rules, Cut elimination implies decidability, as all rules but (Cut) strictly decrease the size of a sequence in terms of some simplicity metric.

The Gentzen presentation for NL is given in figure 2.9. To extend it to the other Lambek calculi and the multimodal case, the same structural rules can be used that were presented in the previous sections. Note that sequents are now written $\Gamma \Rightarrow C$, and that the rules are labelled according to whether they introduce an operator on the left ((/L), (\L), (• L)) or on the right hand side of the sequent ((/R), (\R), (• R)).

To prove Cut elimination for NL [25, p. 109], one shows that a proof containing (Cut) rules can be transformed into a Cut-free proof by subsequently pushing the applications of (Cut) upwards in the proof tree. In an application of (Cut) at the top of the proof tree, one of the premises is an axiom, and the

$$\text{(Ax)}\ \frac{}{x : A \Rightarrow x : A} \qquad \frac{\Delta \Rightarrow t : A \quad \Gamma, x : A, \Gamma' \Rightarrow u : C}{\Gamma, \Delta, \Gamma' \Rightarrow u[t/x]C}\ \text{(Cut)}$$

$$\text{(/R)}\ \frac{\Delta, x : B \Rightarrow t : A}{\Delta \Rightarrow \lambda x.t : A/B} \qquad \frac{\Delta \Rightarrow t : B \quad \Gamma, x : A, \Gamma' \Rightarrow u : C}{\Gamma, f : A/B, \Delta, \Gamma' \Rightarrow u[f(t)/x] : C}\ \text{(/L)}$$

$$\text{(\R)}\ \frac{x : B, \Delta \Rightarrow t : A}{\Delta \Rightarrow \lambda x.t : B\backslash A} \qquad \frac{\Delta \Rightarrow t : B \quad \Gamma, x : A, \Gamma' \Rightarrow u : C}{\Gamma, \Delta, f : B\backslash A, \Gamma' \Rightarrow u[f(t)/x] : C}\ \text{(\L)}$$

$$\text{(•R)}\ \frac{\Delta \Rightarrow t : A \quad \Delta' \Rightarrow u : B}{\Delta, \Delta' \Rightarrow \langle t, u \rangle : A \bullet B} \qquad \frac{\Gamma, x : A, y : B, \Gamma' \Rightarrow t : C}{\Gamma, z : A \bullet B, \Gamma' \Rightarrow t[\pi_1(z)/x, \pi_2(z)/y] : C}\ \text{(•L)}$$

Figure 2.10: L: Sugared Gentzen presentation with λ-terms [25, pp. 108, 117]

other premise will be identical to the conclusion of the (Cut), so that the rule application can be discarded.

Normal form parsing

If as the goal of parsing, one considers to obtain a representation of the semantics of the sentence via the Curry–Howard isomorphism, then the Gentzen presentation is still not optimal for proof search.

Consider the associative Lambek calculus L, whose Gentzen rules, annotated with λ-terms and with (Ass1) and (Ass2) "compiled in" by using a comma instead of the product operator, are given in figure 2.10. (Note that Γ and Γ' represent possibly empty sequences of categories.) Like CCG, L suffers from the "spurious ambiguities" problem, as demonstrated in figure 2.11.

To avoid "spurious ambiguities", König [18] put the search for *normal form proofs* on the agenda of CTL parsing: If one would be able to construct an equivalence relation that regards two proofs as equivalent if they yield the same semantics, then one could try to restrict parsers for CTL to produce only derivations corresponding to the representatives of that relation. These representatives could then be considered as the normal forms of the class of proofs they are representing.

$$\dfrac{np \Rightarrow np \qquad \dfrac{np \Rightarrow np \quad s \Rightarrow s}{np, np\backslash s \Rightarrow s}\,(\backslash\mathrm{L})}{np, (np\backslash s)/np, np \Rightarrow s}\,(/\mathrm{L})$$

(a) Dan (likes parsnips)

$$\dfrac{\dfrac{\dfrac{np \Rightarrow np \quad s \Rightarrow s}{np, np\backslash s \Rightarrow s}\,(\backslash\mathrm{L})}{np \Rightarrow s/(np\backslash s)}\,(/\mathrm{R}) \qquad \dfrac{np \Rightarrow np \qquad \dfrac{np\backslash s \Rightarrow np\backslash s \quad s \Rightarrow s}{s/(np\backslash s), np\backslash s \Rightarrow s}\,(/\mathrm{L})}{s/(np\backslash s), (np\backslash s)/np, np \Rightarrow s}\,(/\mathrm{L})}{np, (np\backslash s)/np, np \Rightarrow s}\,(\textsc{Cut})$$

(b) (Dan likes) parsnips

Figure 2.11: "Spurious ambiguities" in L

The semantic criterion for normal forms was formalised by Hepple [13], who shows that $\beta\eta$-equivalence is an equivalence relation in the sense of König.[2] This result is partially suggested by Cut elimination, which, through the Curry–Howard correspondence, is strongly linked to β-reduction: It turns out that the semantics of two proofs that are equivalent modulo Cut elimination are equivalent modulo β-reduction. To illustrate this point, consider the λ-terms associated to the proofs in figure 2.11, given in figure 2.12. While the λ-term for the Cut-free proof in figure 2.12(a) is completely β-reduced, the term for the proof using (Cut) (figure 2.12(b)) is not.

A calculus that only allows derivations whose proof-terms are $\beta\eta$-reduced is given in figure 2.13 [12, 25]. Instead of trying to derive the sequent $\Gamma \Rightarrow C$, this calculus tries to derive $\Gamma \Rightarrow C^{\star}$, where $\star$ is a control label on categories that governs the sequence of inference rule applications. It forces the derivation to first apply all right-rules, then apply the "mode-switching" rule ($\star$R), and then apply all left-rules. The resulting proofs are in $\beta\eta$-normal form [13].

[2] Two λ-terms t_1, t_2 are equivalent modulo $\beta\eta$-reduction, $t_1 \equiv_{\beta\eta} t_2$, if they can both be reduced to the same term t, using the following reduction rules: $(\lambda x.t)(u) \rightarrow_{\beta\eta} t[u/x]$ (β-reduction), $\lambda x.f(x) \rightarrow_{\beta\eta} f$ (η-reduction).

$$\dfrac{w : np \Rightarrow w : np \qquad \dfrac{u : np \Rightarrow u : np \quad v : s \Rightarrow v : s}{u : np, f : np\backslash s \Rightarrow f(u) : s}\,(\backslash\mathrm{L})}{u : np, g : (np\backslash s)/np, w : np \Rightarrow (g(w))(u) : s}\,(/\mathrm{L})$$

(a) Dan (likes parsnips)

$$\dfrac{\dfrac{\dfrac{u : np \Rightarrow u : np \quad v : s \Rightarrow v : s}{u : np, f : np\backslash s \Rightarrow f(u) : s}}{u : np \Rightarrow \lambda f.f(u) : s/(np\backslash s)}\,(/\mathrm{R}) \qquad \dfrac{x : np \Rightarrow x : np \qquad \dfrac{g : np\backslash s \Rightarrow g : np\backslash s \quad w : s \Rightarrow w : s}{h : s/(np\backslash s), g : np\backslash s \Rightarrow h(g) : s}}{h : s/(np\backslash s), i : (np\backslash s)/np, x : np \Rightarrow h(i(x)) : s}}{u : np, i : (np\backslash s)/np, x : np \Rightarrow (\lambda f.f(u))(i(x)) : s}\,(\textsc{Cut})$$

(b) (Dan likes) parsnips

Figure 2.12: Lambda terms for the "spurious ambiguities" in figure 2.11

$$(\mathrm{Ax}/\star\mathrm{L})\ \dfrac{}{x : p^\star \Rightarrow x : p} \qquad \dfrac{\Gamma, u : B^\star, \Gamma' \Rightarrow t : p}{\Gamma, u : B, \Gamma' \Rightarrow t : p^\star}\,(\star\mathrm{R})$$

$$(/\mathrm{R})\ \dfrac{\Delta, x : B \Rightarrow t : A^\star}{\Delta \Rightarrow \lambda x.t : A/B^\star} \qquad \dfrac{\Delta \Rightarrow u : B^\star \quad \Gamma, x : A^\star, \Gamma' \Rightarrow t : C}{\Gamma, s : A/B^\star, \Delta, \Gamma' \Rightarrow t[s(u)/x] : C}\,(/\mathrm{L})$$

$$(\backslash\mathrm{R})\ \dfrac{x : B, \Delta \Rightarrow t : A^\star}{\Delta \Rightarrow \lambda x.t : B\backslash A^\star} \qquad \dfrac{\Delta \Rightarrow u : B^\star \quad \Gamma, x : A^\star, \Gamma' \Rightarrow t : C}{\Gamma, \Delta, s : B\backslash A^\star, \Gamma' \Rightarrow t[s(u)/x] : C}\,(\backslash\mathrm{L})$$

Figure 2.13: Hendriks' $\mathsf{L}^\star$ calculus [25, p. 164]

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[Z] \vdash [Z] \quad Z\backslash W \vdash Z\backslash W}{([Z], Z\backslash W) \vdash W}\,(\backslash\mathrm{E}) \qquad W\backslash Y \vdash W\backslash Y}{(([Z], Z\backslash W), W\backslash Y) \vdash Y}\,(\backslash\mathrm{E})}{([Z], (Z\backslash W, W\backslash Y)) \vdash Y}\,(\mathrm{Ass1})}{(Z\backslash W, W\backslash Y) \vdash Z\backslash Y}\,(\backslash\mathrm{I}) \qquad (Z\backslash Y)\backslash X \vdash (Z\backslash Y)\backslash X}{((Z\backslash W, W\backslash Y), (Z\backslash Y)\backslash X) \vdash X}\,(\backslash\mathrm{E})$$

(a) with hypothetical reasoning

$$\dfrac{\dfrac{\dfrac{Z \vdash Z \quad Z\backslash W \vdash Z\backslash W}{(Z, Z\backslash W) \vdash W}\,(\backslash\mathrm{E}) \qquad W\backslash Y \vdash W\backslash Y}{((Z, Z\backslash W), W\backslash Y) \vdash Y}\,(\backslash\mathrm{E}) \qquad Y\backslash X \vdash Y\backslash X}{(((Z, Z\backslash W), W\backslash Y), X) \vdash X}\,(\backslash\mathrm{E})$$

(b) without hypothetical reasoning

Figure 2.14: Example for Hepple's compilation method [cf. 15]

Hypothetical reasoning

Besides the "spurious ambiguities", another problem in parsing Lambek grammars is the proper treatment of the (I) rules of the Natural Deduction presentation (figure 2.3). Figure 2.14(a) shows a derivation in L in which an assumption Z (marked with square brackets) is used to derive the category Y, but then discharged by the ($\backslash$I) rule. Traditional chart-based approaches to parsing have difficulties with such *hypothetical reasoning*, as the hypothetical Z category does not find a place on a linearly ordered chart.

Hepple [15] suggests a compilation method, in which higher-order assumptions (such as $(Z\backslash Y)\backslash X$ in figure 2.14(a)) are split up into a hypothetical (Z) and a first-order residue ($Y\backslash X$), leading to (I)-free derivations (figure 2.14(b)). This procedure requires an indexing regime that ensures that each excised hypothetical must be used to derive the ticket of its associated residue, and a modified form of β-reduction. The result of the compilation can be parsed by an Earley-style predictive chart algorithm.

Proof nets

Moot [27] shows how the $\mathsf{NL}\Diamond_{\mathcal{R}}$ calculus (NL amended by modes, unary operators, and an arbitrary set of structural rules $\mathcal{R}$) can be processed using *proof nets*, a graph-based proof theory originally developed for linear logic. Sequents are translated into proof nets, for which a graph-theoretic criterion can be formulated that is satisfied if and only if the input sequent is derivable.

Discussion: The complexity of CTL

This section has presented solutions to various problems related to the efficient processing of CTL. From a practical point of view, the ultimate aim would be to obtain polynomial time parsing algorithms, as they are available for other grammar formalisms, such as CCG or Tree Adjoining Grammar (TAG). Unfortunately, for the linguistically interesting calculi discussed here, this goal is either completely out of reach, or at least uncertain.

At the lower end of the scale, the actual complexity of the parsing problems for NL and L remains unknown. The Lambek–van Benthem calculus LP has been shown to be NP complete by Kanovich [27, p. 154] – the formalism resulting from Hepple's compilation method for L that was discussed earlier on can be parsed in polynomial time, but the compilation itself has exponential complexity [15]. In the multimodal case, it is straightforward to show that CTL with an unrestricted set of structural rules (Moot's $\mathsf{NL}\Diamond_{\mathcal{R}}$) are undecidable [5]. If the structural rules are restricted to linear rules, that is, rules that do not introduce new structure into the proof, then the resulting CTL are still PSPACE complete, and their parsing problem is equivalent to that of general context-sensitive languages [27].

The complexity problem can be addressed in at least two ways.

First, one could try to identify *fragments* of $\mathsf{NL}\Diamond_{\mathcal{R}}$ that can be parsed efficiently. One such fragment, simulating the rules of CCG, has been proposed

by Kruijff and Baldridge [20]. The derivational capacity of this fragment is slightly above that of CCG, although its generative capacity is probably the same (Jason Baldridge, p. c.). Moot presents a fragment of $\mathsf{NL}\Diamond_{\mathcal{R}}$ that is strongly equivalent to TAG [27].

The search for manageable fragments is a walk on a thin line, as processing efficiency and linguistic plausibility do not necessarily coincide. Note, furthermore, that establishing equivalence results between fragments of CTL and formalisms with efficient parsing algorithms does not necessarily imply the applicability of these algorithms to CTL. The associative Lambek calculus L for example has been shown to be weakly equivalent to context-free grammars [29] – but up to the present day, no polynomial translation from L into context-free grammar has been found.

The second way in which to handle the efficiency problem is to ignore the theoretical complexity results and to investigate the *practical* behaviour of CTL fragments. This can be a very viable thing to do, considering that formalisms such as Head-driven Phrase Structure Grammar (HPSG), which have been successfully put to practice, are undecidable in the general case.

The practical evaluation of CTL grammar fragments requires software in which (a) new parsers can be assembled and re-assembled quickly, and (b) general rather than formalism-specific techniques are used to tame the potential complexity of the parsing problem induced by a new fragment. These requirements were the main motivations for the application of constraint programming to the parsing of CTL explored in this project.

Chapter 3

Constraint Programming

For many combinatorial problems, no efficient specialised algorithms exist, and one has to fall back on generic techniques to solve them. One sufficient but naïve method is to generate all possible combinations of values and to check, for each combination, if it constitutes a solution to the problem. This approach, which is known as "brute-force search" or "generate and test", cannot avoid the combinatorial explosion that goes along with NP-hard and highly polynomial problems. In the presence of large search spaces (as they usually occur in real-life applications), it is infeasible even for problems with small polynomial complexity.

This chapter introduces constraint programming [4, 23] as an alternative to the "generate and test" paradigm. Constraint programming tries to develop efficient techniques and expressive programming metaphors to formulate and solve *constraint satisfaction problems (CSPs)* (section 3.1). One of its central techniques is *constraint propagation* (section 3.2), which can be regarded as an application of abstract interpretation. In conjunction with domain-independent *constraint services* like *distribution* and *search*, constraint propagation can yield efficient solvers for combinatorial problems (section 3.3).

3.1 Constraint satisfaction

The different formalisations of constraint satisfaction problems that can be found in the literature may be separated into two classes, depending on the view that they take on constraints. *Intensional* definitions, like the one

given here, emphasise the relation between constraint satisfaction and logic by viewing constraints as formulae. In contrast, *extensional* definitions [2, 24] consider constraints as sets of potential partial problem solutions. The extensional view on constraints stresses the combinatorial complexity of constraint satisfaction: Solving a CSPs amounts to generating all consistent combinations of partial solutions.

Definition 3.1 (CSP) A *constraint satisfaction problem CSP* is given by a triple $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$, consisting of a set $\mathcal{V} := \{x_1, \ldots, x_n\}$ of *problem variables*, a set $\mathcal{D} := \{D_1, \ldots, D_n\}$ of associated domains, and a set $\mathcal{C}$ of constraints. A *domain* $D_i \in \mathcal{D}$ is the set of values that the variable x_i ranges over. A *constraint* $C \in \mathcal{C}$ is a logic formula in some fixed first-order *constraint language* such that all its free variables are problem variables.

The present definition leaves implicit a mapping $\textsc{dom} \in \mathcal{V} \to \mathcal{D}$ that associates problem variables with their domains[1] by assuming that they share the same index.

To formalise the notion of a solution to a CSP, some notation is needed: Borrowing from Apt [2], a *scheme on n* is supposed to mean an increasing sequence $\mathfrak{s} := s_1, \ldots, s_k$ of different elements from $[1, n]$. Given an n-tuple $\vec{v} := (v_1, \ldots, v_n)$ and a scheme $\mathfrak{s}$ on n, let $\vec{v}\,[\mathfrak{s}]$ denote the k-tuple $(v_{s_1}, \ldots, v_{s_k})$. A constraint $C \in \mathcal{C}$ is said to be *with scheme* $\mathfrak{s}$, if $\{x_{s_1}, \ldots, x_{s_k}\}$ are its free variables. Finally, if a constraint C is with scheme $\mathfrak{s}$, the notation $C\ \vec{v}\,[\mathfrak{s}]$ refers to C with the relevant values of $\vec{v}$ substituted for its parameter variables.

Definition 3.2 (Solution to a CSP) A *solution* to a CSP $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ is a tuple $\vec{s} \in D_1 \times \cdots \times D_n$, such that for every constraint $C \in \mathcal{C}$, if C is with scheme $\mathfrak{s}$, $C\ \vec{s}\,[\mathfrak{s}]$ holds with respect to the intended interpretation of the constraint language.

[1] In the terminology of Müller [24], this mapping is called *domain store*.

Example: Send More Money

As an example for a CSP, consider the following puzzle, which, according to Knuth [19, p. 6], was first published in 1924 by H. E. Dudeney in volume 68 of the *Strand* magazine. The task is to find a different decimal digit for every letter, such that the following represents a correct sum:

$$\begin{array}{cccccc} & \texttt{S} & \texttt{E} & \texttt{N} & \texttt{D} \\ + & \texttt{M} & \texttt{O} & \texttt{R} & \texttt{E} \\ \hline \texttt{M} & \texttt{O} & \texttt{N} & \texttt{E} & \texttt{Y} \end{array}$$

As a side condition, the leading digits of every row must not be zero.

In terms of definition 3.1, the problem can be phrased as follows: The set of problem variables is $\{s, e, n, d, m, o, r, y\}$. Each of them ranges over the same domain, the set of decimal digits. The constraints are as follows:

- The values of the variables have to be pairwise distinct.
- Neither `S` nor `M` must be assigned the value 0: $s \neq 0 \wedge m \neq 0$.
- The values of the variables must fit the sum:

$$\begin{aligned} 1000 \times s + 100 \times e + 10 \times n + d + \\ 1000 \times m + 100 \times o + 10 \times r + e = \\ 10000 \times m + 1000 \times o + 100 \times n + 10 \times e + y. \end{aligned}$$

The constraint language used in this problem is the linear arithmetic over (a finite subset of) $\mathbb{N}$, together with its standard interpretation. The problem has a unique solution, which the reader is encouraged to puzzle out.[2]

[2] Incidentally, the digits in the solution to this puzzle can be arranged to form the date on which the United States of America passed the Alien Act, allowing the president to deport dangerous aliens.

Backtracking

What are efficient ways of obtaining one, all, or some solutions of a CSP?

The naïve "generate and test" approach suffers from the deficiency that generation is uninformed: The information contained in the constraints is not used *before* a new value assignment is generated.

Backtracking is a refinement of "generate and test", in which a constraint with scheme $\mathfrak{s}$ is checked as soon as its parameter variables have been instantiated. For example, if the problem variables are $\{x, y, z\}$, and x and y have been instantiated to values a and b, all constraints with scheme x, y can be checked before instantiating z. If any of them fails, $x = a, y = b$ is no longer a potential partial solution, and the algorithm can backtrack to the point where it made the last instantiation for y. Backtracking is a significant improvement over "generate and test", as it might prune the search space considerably. Its major deficiency is the fact that conflicts in the assignment of values to variables are still not recognised before they actually occur. The next section will present techniques by which this may be achieved.

3.2 Constraint propagation and abstract interpretation

The principal insight of constraint programming is that constraints can be used as first-class computation objects. By analysing the constraints of a problem, one might be able to reduce it to a simpler one, before doing any solution search at all. This process, known as *constraint propagation*, can be interleaved with search to yield an efficient technique for solving a CSP (section 3.3).

Notions of consistency

The effect of constraint propagation is the elimination of *inconsistencies* in the relation between domains and constraints. Different notions of consistency are named with reference to the *constraint graph* of a CSP.

Definition 3.3 (Constraint graph) The *constraint graph* of a CSP $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ is the graph (V, E) such that $V = \mathcal{V}$ and

$$E = \left\{ (x_1, x_2) \in \mathcal{V}^2 \mid \exists C \in \mathcal{C}^\flat \colon C \text{ is with scheme } x_1, x_2 \right\},$$

where $\mathcal{C}^\flat$ refers to the set of binarised constraints from $\mathcal{C}$.[3]

The most fundamental notion of consistency is *node consistency*, which holds if every unary constraint of the CSP is satisfied.

An edge (x_1, x_2) in the constraint graph is *arc consistent*, if for every value v_1 in the domain of x_1 there is a corresponding value v_2 in the domain of x_2 such that each constraint $C \in \mathcal{C}^\flat$ with scheme x_1, x_2 is satisfied. Values v_1 for which this condition is violated can be removed from their corresponding domains, as they cannot possibly be part of any solution to the CSP. The process of excluding inconsistent values is known as *domain reduction*.

A number of algorithms have been proposed for domain reduction. One example, AC-3, is given in figure 3.1. It manages a collection Q of potentially inconsistent edges in the constraint graph G, and for each arc (x_k, x_l) checks if the domains of the corresponding variables can be reduced. If this is the case, all arcs (x_i, x_k) need to be reconsidered, as the removal of a value from the domain of x_k may have rendered them inconsistent. In the case of finite domains, the algorithm terminates because new arcs are only added to the agenda after the domain of one of their variables has been reduced.

Apt [2] develops *chaotic iteration* as a general formal framework for the analysis of propagation algorithms. A chaotic iteration is a sequence of function applications in which each function is applied infinitely often. Apt shows that the outputs of many of the standard constraint propagation algorithms

[3] It can be shown that any CSP can be transformed into an equivalent CSP containing unary and binary constraints only.

```
Revise(x_1, x_2)
1  flag ← false
2  for each v_1 ∈ D_1
3  do if ∀C ∈ 𝒞 ∃v_2 ∈ D_2 : C(v_1, v_2)
4        then skip
5        else  D_1 ← D_1 − {v_1}
6              flag ← true
7  return flag

AC-3(G)
1  Q ← { (x_i, x_j) ∈ arcs(G) | i ≠ j }
2  while Q ≠ ∅
3  do (x_k, x_l) ← any arc from Q
4     Q ← Q − (x_k, x_l)
5     if Revise(x_k, x_l)
6        then Q ← Q ∪ { (x_i, x_k) ∈ arcs(G) | i ≠ k, i ≠ l }
7        else  skip
```

Figure 3.1: AC-3

can be regarded as the fixpoints of chaotic iterations employing *constraint reduction* or *domain reduction* functions. A constraint reduction function reduces the number of potential partial solutions in the extension of a constraint. A domain reduction function reduces the set of possible values for a problem variable.

Domain reduction

Abstracting away from specific propagation algorithms, one can look for general techniques of reasoning about the solutions of CSPs without actually generating them. Consider the following problem:

$$x \in \{1,3\}, y \in \{2,3\}, z \in \{2,3,4,6,9\}, z = x + y$$

When the standard backtracking algorithm is applied to this problem, it will generate all $2 \times 2 \times 5 = 20$ different pairings of values, and check the constraint against each of them. Due to the mathematical properties of the addition operation however, several of the potential value assignments can be recognised as inconsistent with the constraint *before* they are generated in the first place.

Instead of the domains of the problem variables v_i, one can consider the intervals induced by their minimal and maximal elements, written as $[v_i^{\downarrow}, v_i^{\uparrow}]$. As addition on integers is monotonic,

$$[z^{\downarrow}, z^{\uparrow}] = [x^{\downarrow}, x^{\uparrow}] + [y^{\downarrow}, y^{\uparrow}] = [x^{\downarrow} + y^{\downarrow}, x^{\uparrow} + y^{\uparrow}] = [1+2, 3+3] = [3,6].$$

This calculation reveals that the previous domain for z was inconsistent with the $z = x + y$ constraint: z cannot possibly take the values 2 or 9, only values between 3 and 6. Removing the inconsistency yields the new problem

$$x \in \{1,3\}, y \in \{2,3\}, z \in \{3,4,6\}, z = x + y,$$

for which only $2 \times 2 \times 3 = 12$ different value assignments would have to be generated by backtracking. Notice that the new problem has the same set of solutions as the old one.

Abstract interpretation

Looking at pairs of boundary values such as $[z^{\downarrow}, z^{\uparrow}]$ instead of the set of values *within* the boundaries is a typical example for *abstract interpretation* [6]. Abstract interpretation was originally devised for the static analysis of computer programs, but can be applied to a wide range of different problems.

A semantics or interpretation of a formal system F can be seen as a mapping $\mathcal{I} \in \mathsf{F} \to \mathfrak{D}$, where $\mathfrak{D}$ is some domain of semantic values – for example, finite sets of program execution states, or higher-order functions, as in the semantics for the Lambda calculus. Abstract interpretation replaces the interpretation mapping $\mathcal{I}$ by a mapping $\mathcal{I}^\sharp \in \mathsf{F} \to \mathfrak{D}^\sharp$, with $\mathfrak{D}^\sharp$ being a (usually finite) "abstract" semantic domain suitable for the problem. The concrete domain and the abstract domain are then linked by an *abstraction function* $\alpha \in \mathfrak{D} \to \mathfrak{D}^\sharp$. Conversely, there also is a *concretisation function* $\gamma \in \mathfrak{D}^\sharp \to \mathfrak{D}$.

In the above example, the concrete domain $\mathfrak{D}$ of sets of integers was replaced by the abstract domain $\mathfrak{D}^\sharp$ of integer intervals. The corresponding abstraction and concretisation functions are straightforward:

$$\alpha(N) = [\min N, \max N], N \subseteq \mathbb{N}, N \neq \emptyset, \qquad \gamma([x,y]) = \{\, z \mid x \leq z \leq y \,\}.$$

The abstraction is sound in the sense that, if $+^\sharp$ is the addition on intervals,

$$\alpha(X) +^\sharp \alpha(Y) = \alpha(Z) \quad \Rightarrow \quad Z \subseteq \gamma(\alpha(Z)),$$

where X and Y are the domains of x and y, respectively; this follows from the monotonicity of addition. (Notice that if both X and Y are convex, then so is Z, and $\subseteq$ can be replaced by $=$ on the right hand side of the rule.)

To obtain a possibly reduced domain Z', we can therefore intersect Z with $\gamma(\alpha(Z))$. This technique is one instance of a general propagation paradigm called *bounds propagation*.

Depending on the abstract domains, computations in $\mathcal{D}^\sharp$ can be considerably more efficient than computations in $\mathcal{D}$. This was the case in the example: The interval $[z^\downarrow, z^\uparrow]$ can be computed more easily from the intervals $[x^\downarrow, x^\uparrow]$ and $[y^\downarrow, y^\uparrow]$ than the set Z of all possible sums of values from X and Y. In spite of its relative computational inexpensiveness, bounds propagation is quite expressive, however: Schulte and Stuckey show that in many situations, more expensive propagation methods can be replaced by bounds propagation without increasing the search space for the CSP [32].

The benefits of abstract interpretation do not come without a price, though: While the abstraction is required to be sound, it will not usually be complete – every answer obtained from abstract interpretation is correct, but not all questions can be answered. With regard to interval abstraction, a set of integers cannot in general be replaced by its boundary elements without loosing information: Two sets with the same boundaries need not necessarily be equal. More formally, the concretisation function γ is not in general inversely related to the abstraction function α.

In spite of this, abstract interpretation is a very powerful and well-founded technique. Furthermore, in the context of constraint programming, the expressiveness of the constraint language in principle allows the simultaneous application of different abstractions, which may be complete in combination, or could at least lead to a strong propagation that only requires little search.

3.3 Propagation, distribution, and search

The present section deals with the question how constraint technology can be integrated into an existing programming language. In its exposition it follows the framework of Schulte [31], who develops *computation spaces* as

New: *Script* → *Space*	creates a new space from a script
Inject: *Space* × *Script* → *Space*	injects an additional script into a space
Merge: *Space* × *Space* → *Space*	merges two spaces
Ask: *Space* → *Status*	asks the status of a space
Clone: *Space* → *Space*	clones a space
Commit: *Space* × `int` → *Space*	selects a daughter of a distributable space

Figure 3.2: Operations on spaces [31, section 4.8]

the central metaphors for constraint programming and shows that they are seamlessly integrated into the Oz Programming Model [33], which has been implemented in the Mozart/Oz system [28].

Computation spaces

Computation spaces are first-class programming objects providing conceptual abstractions from the internals of a CSP and concrete propagation algorithms. The possible operations on computation spaces are given in figure 3.2.

Definition 3.4 (Computation space) A *computation space* is composed of a *constraint store*, which holds information about the currently possible values for the problem variables, and a set of *propagators*. A propagator is a concurrent computational agent that contributes new information to the constraint store.

The information in the constraint store is expressed in terms of a conjunction of *basic constraints*. The set of basic constraints available in a computation space will typically depend on a number of factors, such as the programming language used, the problem domain, and the efficiency with which the basic constraints can be implemented. For problems involving finite sets of objects, a natural choice would be to have only one basic constraint, $x \in D$.

Propagation

In order to express more complicated relations between variables, *non-basic constraints* are employed. For example, in the context of finite domain integer problems, natural non-basic constraints are $x < y$ or $\textsc{different}(x_1, \ldots, x_n)$.

Non-basic constraints are implemented by propagators, which act as translators into the language of the basic constraints. A propagator implementing a non-basic constraint ψ can insert a new basic constraint β into the constraint store ϕ if and only if this new constraint is *adequate* ($\phi \wedge \psi \Rightarrow \beta$), *new* ($\phi \not\Rightarrow \beta$), and *consistent* ($\phi \wedge \beta \not\Rightarrow \bot$). If $\phi \Rightarrow \psi$, the propagator implementing ψ is called *entailed*, and ceases to exist. If ψ is inconsistent with ϕ, its propagator is *failed*.

A computation space is called *stable*, if its propagators cannot add any more information to its constraint store. A stable space containing a failed propagator is itself called *failed*, as there can be no solution for the problem that it encapsulates. A stable space that does not contain any propagators is called *solved* or *succeeded*.

Computation spaces provide an interface for concrete propagation algorithms: The implementation of propagators becomes an orthogonal issue that depends on the architecture of the underlying programming language. Müller [24] discusses the implementation of propagators in Mozart/Oz.

Distribution

Constraint propagation is no complete solution strategy for a CSP: A space may become stable without being either solved or failed. Consider the following example:

$$x \in \{1, 2\}, y \in \{1, 2\}, x \leq y.$$

Here, the propagator implementing the $\leq$-constraint is not able to add any new domain-reducing information to the constraint store, but is not entailed, either. A stable but unsolved space is called *distributable*.

Distribution, which is also known as labelling or branching, is a way of non-deterministically reducing the size of a constraint problem. To distribute a computation space S with respect to a constraint β, S is *cloned*, resulting in two spaces S_1 and S_2, whereupon β is injected into S_1, and $\neg\beta$ into S_2. Constraint propagation can then continue in both daughter spaces. Which constraint β is chosen for distribution is a matter of heuristics: distributable spaces are *choice points* in the solution of the problem.

An obvious distribution strategy is to reduce the size of some domain D by picking an arbitrary element d from D and injecting $d \in D$ into one space and $d \notin D$ into the other. As a refinement, one might want to pick the *smallest* domain D, in the hope that propagation is stronger in smaller domains. The first strategy is known as *naïve* distribution, the second as *first fail*. More advanced distribution strategies base their decision on the result of a cost function, which valuates a distributable space according to some measure in order to decide which new constraints to inject.

Search

After a space has been distributed, it remains an orthogonal issue as to which of its daughter spaces propagation shall proceed in first. This is determined by a *search strategy*, which is implemented in a *search engine*. An overview of standard search strategies can be found in standard AI textbooks [e. g., 30].

As an example for a search engine, an algorithm for simple depth-first exploration (DFE) of the solutions of a computation space S is given in figure 3.3. DFE finds the first solved space for the problem encapsulated in S, if any. The function Ask returns the status of a space, which can be either failed, succeeded, or distributable. In the latter case, the distribution strategy will determine two potential daughter spaces. Before the constraint solver Commits to the first daughter by injecting the appropriate constraints, S is

```
DFE(S)
 1  status ← Ask(S)
 2  switch
 3     case status = failed :
 4           return none
 5     case status = succeeded :
 6           return S
 7     case status = daughters :
 8           C ← Clone(S)
 9           Commit(S, 1)
10           status′ ← DFE(S)
11           switch
12              case status′ = none :
13                    Commit(C, 2)
14                    return DFE(C)
15              case status′ = T :
16                    return T
```

Figure 3.3: Depth-first exploration [31, p. 45]

Cloned, so that the second daughter space can be recovered in case the search in the first one should not yield any solutions.

Constraint services

Distribution and search are two examples of what Schulte calls *domain-independent constraint services* – they do not rely on the concrete application area, and one can imagine whole "factories" of general-purpose distribution and search engines to be made available as libraries to a programming language supporting the concept of computation spaces. Propagation, in contrast, is a *domain-dependent* constraint service: Different groups of problems need different propagation algorithms.

The interaction between the various constraint services is illustrated well by *best-solution search* [31, section 6]. The objective of best-solution search is to exhaustively explore the search space for a problem to eventually find the best solution to that problem, according to some rating function. Brute-force search cannot in general be applied to this problem, as the search space may be immense. In the computation space framework, however, it is easily possible to prune the search space by insisting, after finding a solution, that all further solutions must be better than that last one. This condition can be formulated as a constraint and injected into the problem space, which may lead to stronger propagation, eliminating unnecessary branches of the search tree.

Interactive tools

The task of assembling efficient solvers for constraint satisfaction problems is of a highly experimental nature. Both the design and selection of domain-specific propagators and the choice of a distribution and search strategy cannot be considered independently from concrete problem instances. For a constraint programming system, the availability of interactive development tools therefore is essential.

The Mozart/Oz system [28] provides a range of such tools. One of the most important ones is the Oz Explorer [31, chapter 8], with which the search trees of CSPs can be displayed, and the status of the computation spaces involved can be inspected. This enables the developer to see which additional constraints may be imposed to increase propagation and decrease the need for search, and if the distribution and search strategies can be improved.

To give an example, figure 3.4 shows the search tree for the "Send More Money" problem from section 3.1, as returned from the "Oz Explorer". Different kinds of computation spaces are represented by different symbols: Circles ● mark distributable, squares ■ failed, and diamonds ◆ solved spaces.

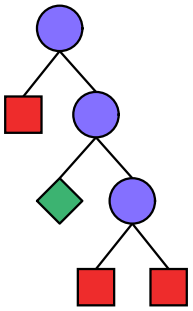


Figure 3.4: Search tree for the "Send More Money" problem

The distribution strategy applied here is "first fail"; the search strategy is the default exhaustive depth-first search. Note that constraint propagation reduces the search problem, which normally would involve almost 100,000,000 choices, to 4 choices.

Discussion: The benefits of constraint programming

Constraint programming has become increasingly popular over the last few years. This is due to three main reasons:

First, constraint programming is inherently declarative, and thus comes very close to the ideal programming paradigm in which the programmer merely states the requirements of the problem (in form of constraints), and the programming language does all the rest [31].

Second, constraint programming enables the rapid prototyping of solutions to problems for which no specialised algorithms exist or would be worth implementing.

Finally, more and more programming languages support constraint programming, either in form of libraries, or from within the core system (as in Mozart/Oz). Propagation, distribution and search can be combined with other features of the language to yield powerful applications.

These three features of constraint programming make it an ideal candidate for the application to Categorial Type Logics: Rapid prototyping was one

of the requirements for the CTL development tools mentioned at the end of chapter 2, and as the next chapter will show, the declarative nature of constraint programming allows an almost immediate translation from the logical and structural properties of CTL into the constraint language.

Chapter 4

Configuring Categorial Grammars

This chapter presents the principal results of this thesis. In section 4.2, a novel formulation of the parsing problem for categorial grammars is given that makes this task accessible to the application of constraint programming. Section 4.3 demonstrates the usability of the new framework by integrating into it a recent proposal for constraints on structural control [11].

4.1 Parsing as tree configuration

The result of a parse is a finite tree composed according to the rules of a grammar. Parsing can therefore be viewed as a *tree configuration problem*: The positions of components of the tree constitute the problem variables of a constraint satisfaction problem, the composition principles form the constraints.

Tree regions

Tree regions are a powerful concept to express the well-formedness conditions of a tree configuration problem in a declarative way [10]: The position of each node N of a tree is specified relative to a number of disjoint sets of other nodes. For example, the region N_{up} would contain all nodes that lie strictly above N on the path from N to the root node of the tree. A collection of tree regions is *complete*, if it partitions the set of all nodes in the tree. A node can then be uniquely identified by the tree regions associated to it. One example for a complete set of tree regions is shown in figure 4.1.

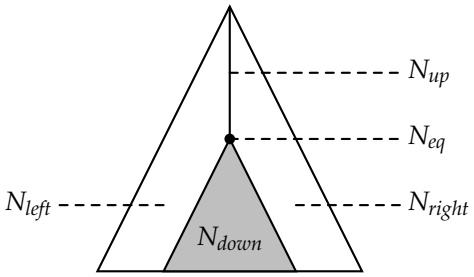


Figure 4.1: Tree regions

Set constraints

If a node is identified by its associated tree regions, imposing restrictions on the position of the node amounts to imposing restrictions on sets of nodes. For example, writing *Nodes* for the set of all nodes in a tree, the completeness condition can be written as

$$\forall N \in Nodes\colon Nodes = N_{eq} \uplus N_{up} \uplus N_{down} \uplus N_{side}. \tag{4.1}$$

In the context of constraint programming, restrictions like this are most naturally expressed in terms of *set constraints* [10]. The declarative semantics of a set constraint is its standard mathematical reading. Operationally, it is implemented by a propagator as described in section 3.3. Constraint propagation translates set constraints into basic constraints delimiting the lower and upper bounds of a set [9]. For example, the constraint $S_1 \subseteq S_2$ can be translated into the two basic constraints $\lfloor S_1 \rfloor \subseteq S_2$ and $S_1 \subseteq \lceil S_2 \rceil$, where $\lfloor S_1 \rfloor$ and $\lceil S_1 \rceil$ are the most specific lower and upper bounds for S_1 and S_2 currently entailed by the constraint store:

$$\lfloor S_1 \rfloor = \bigcup \{\, D \mid D \subseteq S \,\} \qquad \lceil S_2 \rceil = \bigcap \{\, D \mid S \subseteq D \,\}$$

Constraint propagators for all of the standard set relations are implemented in the Finite Set library of the Mozart/Oz System.

Constraints on trees

Using set constraints on tree regions, it is possible to state a number of well-formedness criteria, which, taken together, will permit only tree structures as solutions to the configuration problem. These criteria can all be derived from three elementary graph-theoretic properties of trees:

1. Each node in a tree has at most one incoming edge.
2. There is exactly one node (the root) with no incoming edges.
3. There are no cycles.

To state the first property, two new tree regions containing the mothers and daughters of a node are introduced, and the cardinality of the set of mothers is constrained. The two regions are dual in the sense that if a node M is a daughter of another node N, then N must be M's mother.

$$\forall N \in \mathit{Nodes}\colon 0 \leq |N_{\mathit{mothers}}| \leq 1 \tag{4.2}$$

$$\forall M \in \mathit{Nodes}\colon \forall N \in \mathit{Nodes}\colon M \in N_{\mathit{daughters}} \iff N \in M_{\mathit{mothers}} \tag{4.3}$$

Condition 4.3 can be implemented by a propagator for a *reified* constraint: When a constraint C is reified by an integer variable $I \in \{0, 1\}$, I takes the value 1 if and only if C is satisfied. If we write $[\![\cdot]\!]$ for the value of the variable reifying a constraint, equation 4.3 could then be re-stated as

$$\forall M \in \mathit{Nodes}\colon \forall N \in \mathit{Nodes}\colon [\![M \in N_{\mathit{daughters}}]\!] = [\![N \in M_{\mathit{mothers}}]\!] \tag{4.3$'$}$$

The sets of mothers and daughters of a node are linked with the other tree regions such that the set of nodes strictly above a node N (N_{up}) is the union of the sets of nodes weakly above any mother of N. Dually, the set of nodes strictly below a node N (N_{down}) is the union of the sets of nodes

weakly below any daughter of N. To formulate this condition, another pair of additional tree regions N_{equp} and N_{eqdown} is defined such that

$$N_{equp} = N_{eq} \uplus N_{up} \tag{4.4}$$

$$N_{eqdown} = N_{eq} \uplus N_{down} \tag{4.5}$$

$$N_{eq} = N_{equp} \cap N_{eqdown} \tag{4.6}$$

The linking constraints can then be stated in terms of *selection constraints*:

$$N_{up} = \bigcup \{ M_{equp} \mid M \in N_{mothers} \} \tag{4.7}$$

$$N_{down} = \bigcup \{ M_{eqdown} \mid M \in N_{daughters} \} \tag{4.8}$$

Selection constraints were developed by Duchier [8, 9]. They have been successfully used to model selectional ambiguity phenomena such as lexical ambiguity, and will also be employed for the unfolding encoding developed in section 4.2. Propagators for selection constraints are available as an add-on package for Mozart/Oz.

To ensure the second tree property, one needs to make reference to the set of the roots of a tree, $Roots \subseteq Nodes$:

$$|Roots| = 1 \tag{4.9}$$

$$\forall N \in Nodes\colon N \in Roots \iff |N_{mothers}| = 0 \tag{4.10}$$

Every node must be either a root, or the daughter of another node:

$$Nodes = \biguplus \{ N_{daughters} \mid N \in Nodes \} \uplus Roots \tag{4.11}$$

Finally, in conjunction with the other constraints, cycles (property 3) can be disallowed by stating that for each pair (M, N) of nodes, either M and N are the same nodes ($=$), or one lies strictly above the other ($\lhd$), or the two

lie in disjoint segments of the tree ($\|$):

$$\forall M \in \mathit{Nodes}\colon \forall N \in \mathit{Nodes}\colon\ M = N \ \vee\ M \lhd N \ \vee\ N \lhd M \ \vee\ M \parallel N \qquad (4.12)$$

The relation symbols $=$, $\lhd$ and $\|$ are abbreviations for set constraints [10]:

$$M = N \iff M_{eq} = N_{eq}$$
$$M \lhd N \iff N_{eqdown} \subseteq M_{eqdown} \wedge M_{equp} \subseteq N_{equp} \wedge M_{side} \subseteq N_{side} \wedge M \neq N$$
$$M \parallel N \iff M_{eqdown} \subseteq N_{side} \wedge N_{eqdown} \subseteq M_{side}$$

Adding grammatical structure

The conditions given so far are necessary and sufficient to restrict the solutions of a tree configuration problem to *some* tree. The next step towards the formalisation of the parsing problem is to restrict the solutions of the CSP to trees that are licensed by the composition principles of categorial grammar.

Derivations in AB grammar are binary trees. A parse tree using n axioms will therefore always have $2n-1$ nodes, n leaves and $n-1$ inner nodes. To formulate the parsing problem in terms of a CSP, a straightforward approach would be to associate a category label with each of these nodes such that a node is either (a) labelled with the category of a lexical entry for a word in the input sentence or (b) labelled with a category that is the result of the application of an (\E) or (/E) rule to the categories of the nodes of its two daughters.

Unfortunately, a formulation like this leads to relatively weak constraint propagation, as it contains a number of symmetries that allow several solutions where only one is intended. Most obviously, the relative order of the inner nodes is not fixed, so that each grammatical sentence will have $(n-1)!$ different analyses. This problem can be compared to the "spurious ambiguities" problem of traditional parsing methods for categorial grammar. One solution would be to impose some (arbitrary) order on the inner nodes. An

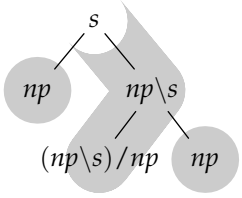


Figure 4.2: Unfoldings partition a proof tree

alternative approach that aims at reducing the total number of nodes will be developed in the next section.

4.2 The unfolding encoding

This section presents the main theoretical result of this thesis, a novel formulation of the parsing problem for categorial grammars, developing and extending an original idea by Denys Duchier (p. c.). It builds on a formal framework [8, 9] that has recently been employed in the implementation of an efficient parser for Topological Dependency Grammar (TDG) [7]. The distinctive feature of the new formulation is that it assumes only *one* node per word in the input sentence, enabling strong constraint propagation.

To illustrate the approach, consider figure 4.2, which shows the proof tree for a simple English sentence (compare the derivation in figure 2.1). Notice that each axiom induces a path towards the root of the tree on which its sub-categories act as functors. The terminal node of such a path is either the root node, or a node that is labelled with a category used as the ticket in the application of an elimination rule. Intuitively, the set of these paths, which will be called the *unfoldings* of their respective axioms, partitions the proof tree into disjoint regions. Because every unfolding corresponds to exactly one axiom, it should therefore be possible to use unfoldings instead of categories as the labels of the nodes in the tree configuration problem. The remainder of this section will formalise this idea.

Proof paths and unfoldings

Proof paths are proof trees that can contain "holes". Holes can be "plugged" with other proof paths to eventually yield proper proof trees.

Definition 4.1 (Proof paths) The set of *AB proof paths* is the smallest set generated by the following inference rules, where A, B, C are categories:

$$\frac{}{C \vdash C}\ (\bot) \qquad \frac{}{\downarrow \vdash C}\ (\downarrow) \qquad \frac{P_1 \vdash B \quad P_2 \vdash B\backslash A}{(P_1 \vdash B) \oslash (P_2 \vdash B\backslash A) \vdash A}\ (\oslash) \qquad \frac{P_1 \vdash A/B \quad P_2 \vdash B}{(P_1 \vdash A/B) \oslash (P_2 \vdash B) \vdash A}\ (\oslash)$$

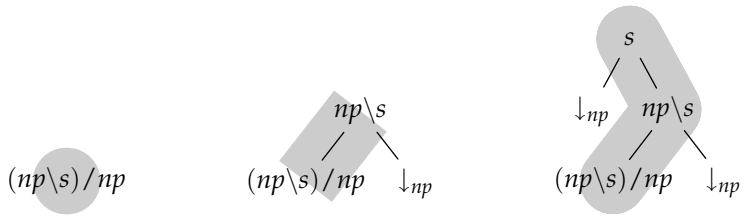
Given a proof path $P \vdash C$, C is called the *head* of P. A proof path of the form $C \vdash C$ is an *anchor*, one of the form $\downarrow \vdash C$ is a *hole*. A proof path is called *proper*, if it contains exactly one anchor. The notation $P[\downarrow \vdash C]$ refers to a proof path P containing a designated hole $\downarrow \vdash C$, and $P[P']$ then refers to the proof path obtained from P by replacing this hole by a new proof path P' with head C. This process will be called *plugging*.

Rules ($\oslash$) and ($\oslash$) express that two proof paths P_1 and P_2 can be composed into a new path P if their heads can be composed according to the ($\backslash$E) and (/E) rules of the Natural Deduction presentation of the Lambek calculus given in figure 2.3. This suggests the following lemma:

Lemma 4.1 Consider a proper proof path $P \vdash C$. If $P' \vdash C$ is the proof path that is obtained from P by plugging all holes $\downarrow \vdash D$ with NL sequents $\Delta \vdash D$, then there is a natural mapping from P' to an NL sequent Γ such that $\Gamma \vdash C$.

Proof: The mapping is as follows:

$$\begin{aligned} [\![\Delta]\!]_{\mathsf{PP}} &= \Delta \\ [\![(P_1 \vdash B) \oslash (P_2 \vdash B\backslash A)]\!]_{\mathsf{PP}} &= ([\![P_1]\!]_{\mathsf{PP}}, [\![P_2]\!]_{\mathsf{PP}}) \\ [\![(P_1 \vdash A/B) \oslash (P_2 \vdash B)]\!]_{\mathsf{PP}} &= ([\![P_1]\!]_{\mathsf{PP}}, [\![P_2]\!]_{\mathsf{PP}}) \end{aligned}$$ (QED)



(a) $d = 0$ (b) $d = 1$ (c) $d = 2$

Figure 4.3: Three unfoldings of a transitive verb

The next definition will establish a designated set of proper proof paths induced by the axioms of a proof tree. These *unfoldings* satisfy the conditions of lemma 4.1 and can therefore be used to build valid proof trees.

Definition 4.2 (Unfoldings) The set of *unfoldings* of a category C is the smallest set generated by the following inference rules:

$$\frac{}{C \vdash C}\ (\bot) \qquad \frac{P \vdash B\backslash A}{(\downarrow \vdash B) \oslash (P \vdash B\backslash A) \vdash A}\ (\oslash) \qquad \frac{P \vdash A/B}{(P \vdash A/B) \oslash (\downarrow \vdash B) \vdash A}\ (\oslash)$$

The same terminology as for proof paths will be used for unfoldings.

To give an example, figure 4.3 shows tree representations of the three unfoldings of the transitive verb category $(np\backslash s)/np$, where holes $\downarrow \vdash C$ are written as $\downarrow_C$. The anchor of these unfoldings is $(np\backslash s)/np \vdash (np\backslash s)/np$. From left to right, the heads are $(np\backslash s)/np$, $np\backslash s$, and s.

Every AB proof tree can be composed of the unfoldings of its axioms:

Lemma 4.2 Let $\Gamma \vdash C$ be any NL derivation using only (\E) and (/E) as its rules. Then a hole-free proof path $P \vdash D$ with $[\![P]\!]_{\mathsf{PP}} = \Gamma$ and $D = C$ can be obtained by choosing for each axiom in the derivation one of its unfoldings and plugging them together.

Proof: Let T be a derivation tree that meets the requirements of the lemma, and use induction over the height of T, h.

If $h = 0$, then T is of shape $C \vdash C$ and has the single axiom C. The only unfolding of C is $C \vdash C$ itself, and $[\![C]\!]_{\mathsf{PP}} = C$.

Now suppose that the lemma holds for every proof tree of height $h < n$, and consider a proof tree of height n. Without loss of generality, assume that the last rule applied in T is ($\backslash$E). Then the top of the tree is of the form

$$\frac{\Gamma \vdash B \quad \Delta \vdash B\backslash A}{(\Gamma, \Delta) \vdash A}\ (\backslash\mathrm{E})$$

By the induction hypothesis, there are two hole-free proof paths $P_\Gamma \vdash B$ and $P_\Delta \vdash B\backslash A$ such that $[\![P_\Gamma]\!]_{\mathsf{PP}} = \Gamma$ and $[\![P_\Delta]\!]_{\mathsf{PP}} = \Delta$. Furthermore, the proof path for the ride category must be the result of the plugging of an unfolding $U = P_\Delta \vdash B\backslash A$ of some axiom X. Then, by rule ($\oslash$), $U' = (\downarrow \vdash B) \oslash (P_\Delta \vdash B\backslash A) \vdash A$ must also be among the unfoldings of X. Extending U to U' and plugging the hole $\downarrow \vdash B$ with the ticket derivation $\Gamma \vdash B$ by lemma 4.1 yields a new hole-free proof path

$$(\Gamma \vdash B) \oslash (P_\Delta \vdash B\backslash A) \vdash A,$$

which satisfies the proof obligation, as

$$[\![(\Gamma \vdash B) \oslash (P_\Delta \vdash B\backslash A)]\!]_{\mathsf{PP}} = ([\![\Gamma]\!]_{\mathsf{PP}}, [\![P_\Delta]\!]_{\mathsf{PP}}) = (\Gamma, \Delta). \qquad \text{(QED)}$$

Lemmata 4.1 and 4.2 proof the initial intuition about unfoldings: Each derivation tree with n axioms can be decomposed into n unfoldings, one for each axiom. In a second attempt, the tree configuration problem can therefore be stated as follows: For each word in the input sentence, (1) select one of its unfoldings, and (2) find a tree that is composed of these unfoldings in a way that is justified by the principles of the grammar. The first task is a problem of selectional ambiguity that can be solved using selection constraints [8]. The second task will be formalised by the notion of the *grid* of a CG.

Relation to partial proof trees

The unfolding encoding can be regarded as a constraint formalisation of the context-free fragment of the *partial proof tree system* of Joshi and Kulick [17]. Partial proof trees (PPTs) essentially are unfoldings that, besides "plugging" (called *application* by Joshi and Kulick), support two other operations, *stretching* and *interpolation*, which increase the generative capacity of PPTs beyond context-freeness. In order to remain in the realm of mildly context-sensitive languages, certain lexical restrictions on the unfoldings of a category have to be imposed on PPTs. For example, an adverb category like $(np\backslash s)\backslash(np^\star\backslash s)$ will not yield the "maximal" unfolding with the holes $\downarrow \vdash (np\backslash s)$ and $\downarrow \vdash np^\star$: The $\downarrow \vdash np^\star$ hole is marked as "not selected" ($\star$). Lexical restrictions like that are unnecessary in the present framework.

Grids

Lemma 4.1 imposes the restriction that two unfoldings shall only be able to plug together when the head of one matches a hole of the other. To formulate this idea in terms of set constraints, the notion of the *grid* of a categorial grammar will be introduced.

It will be useful to arrange some notation that allows switching the perspective between nodes, unfoldings, and the categories of their heads: Letters like M, N will denote nodes. The unfoldings associated to them will be written as U_M, U_N; the categories of their heads as C_M, C_N. Finally, the function UNFOLD is supposed to return the set of unfoldings of a node or category.

The left-to-right ordering of an unfolding's holes induces a structure similar to the subcat list of grammar formalisms like Head-driven Phrase Structure Grammar (HPSG). However, while in HPSG, subcat lists can be regarded as mappings from integers to categories, in the present framework, each unfolding U corresponds to a mapping $\textsc{subcat}(U)$ from pairs of *fields* (left and right in the context of standard categorial grammar) and *positions* within a

field to a *set* of categories. The elements of the domain of this mapping will be referred to as *unfolding indices*. To give an example, the subcat mapping for the unfolding of figure 4.3(c) would be $\{\langle\triangleleft,1\rangle \mapsto \{np\}, \langle\triangleright,1\rangle \mapsto \{np\}\}$, expressing that both the first hole to the left (unfolding index $\langle\triangleleft,1\rangle$) and the first hole to the right ($\langle\triangleright,1\rangle$) of the unfolding's anchor are labelled with the category *np*.

Definition 4.3 (Subcat mapping) Let U be an unfolding with its leaf word being $U_1,\ldots,U_n$. Then the *subcat mapping* of U is the unique solution of

$$\textsc{subcat}(U) = {}^{d_1}_{i_1}[\![U_1]\!]^{d_2}_{i_2} \cup {}^{d_2}_{i_2}[\![U_2]\!]^{d_3}_{i_3} \cup \cdots \cup {}^{d_n}_{i_n}[\![U_n]\!]^{d_{n+1}}_{i_{n+1}},$$

such that

$$\begin{aligned}
{}^{\triangleleft}_{1}[\![\bot \vdash C]\!]^{\triangleright}_{1} &= \varnothing \\
{}^{\triangleleft}_{i+1}[\![\downarrow \vdash C]\!]^{\triangleleft}_{i} &= \{\langle\triangleleft,i\rangle \mapsto \{C\}\} \\
{}^{\triangleright}_{i}[\![\downarrow \vdash C]\!]^{\triangleright}_{i+1} &= \{\langle\triangleright,i\rangle \mapsto \{C\}\}.
\end{aligned}$$

The grid of a categorial grammar is the union of the sets of unfolding indices of all unfoldings possible in that grammar.

Definition 4.4 (Grid) Let $\mathcal{C}$ be a set of categories. The *grid* of $\mathcal{C}$ is defined as

$$\mathcal{G}(\mathcal{C}) = \{\, \mathfrak{I} \mid \exists C \in \mathcal{C} \colon \exists U \in \textsc{unfold}(C) \colon \mathfrak{I} \in \mathrm{dom}(\textsc{subcat}(U)) \,\}.$$

Continuing the example, the set of the three unfoldings in figure 4.3 would have the grid $\{\langle\triangleleft,1\rangle,\langle\triangleright,1\rangle\}$.

The notion of the grid allows the straightforward embedding of the individual $\textsc{subcat}(U)$ mappings for a set $\mathcal{C}$ of categories, which have different domains depending on U, into mappings $\textsc{subcat}_{\mathcal{G}(\mathcal{C})}(U)$ defined on the complete grid. For example, if the unfoldings from figure 4.3 would occur in

a grammar with grid $\{\langle\triangleleft,1\rangle,\langle\triangleright,1\rangle,\langle\triangleright,2\rangle\}$, their respective $\textsc{subcat}_{\mathcal{G}(\mathcal{C})}(U)$ mappings would be

$$\begin{aligned}
4.3(a) &\rightarrow \{\langle\triangleleft,1\rangle \mapsto \emptyset, \langle\triangleright,1\rangle \mapsto \emptyset, \langle\triangleright,2\rangle \mapsto \emptyset\}\,,\\
4.3(b) &\rightarrow \{\langle\triangleleft,1\rangle \mapsto \emptyset, \langle\triangleright,1\rangle \mapsto \{np\}\,, \langle\triangleright,2\rangle \mapsto \emptyset\}\,,\\
4.3(c) &\rightarrow \{\langle\triangleleft,1\rangle \mapsto \{np\}\,, \langle\triangleright,1\rangle \mapsto \{np\}\,, \langle\triangleright,2\rangle \mapsto \emptyset\}\,.
\end{aligned}$$

Linking grids and tree regions

Grids and tree regions can be linked by introducing a function

$$\textsc{plugged} \in \mathit{Nodes} \rightarrow \mathcal{G}(\mathcal{C}) \rightarrow \mathfrak{P}(\mathit{Nodes})$$

that maps a node $N \in \mathit{Nodes}$ and an unfolding index $\mathfrak{I} \in \mathcal{G}(\mathcal{C})$ to the set of those nodes whose associated unfoldings have been plugged into U_N at the hole with the unfolding index $\mathfrak{I}$. As both $\mathcal{G}(\mathcal{C})$ and *Nodes* and are fixed for a given grammar and input sentence, the PLUGGED mapping can be constructed by imposing two set constraints:

First, the set of daughters of a node N is the set of *all* nodes whose associated unfoldings have been plugged into U_N:

$$\forall N \in \mathit{Nodes}\colon N_{\mathit{daughters}} = \bigcup_{\mathfrak{I}} \textsc{plugged}(N)(\mathfrak{I}) \tag{4.13}$$

Second, an unfolding U_M can be plugged into an unfolding U_N at a hole with index $\mathfrak{I}$ if and only if the hole in U_N and the head of U_M are labelled with the same category:

$$\begin{aligned}
&\forall M \in \mathit{Nodes}\colon \forall N \in \mathit{Nodes}\colon \forall \mathfrak{I} \in \mathcal{G}(\mathcal{C})\colon\\
&\qquad (M \in \textsc{plugged}(N)(\mathfrak{I}) \wedge C_M \in \textsc{subcat}(U_N)(\mathfrak{I})) \vee\\
&\qquad\qquad M \notin \textsc{plugged}(N)(\mathfrak{I})
\end{aligned} \tag{4.14}$$

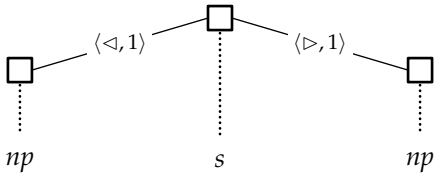


Figure 4.4: A dependency tree for figure 4.3(c)

These last two constraints conclude the presentation of the parsing problem for categorial grammars as a tree configuration problem: Taken together, 4.1–4.14 will permit those and only those tree structures as solutions to a CSP that can be obtained from the words in the input sentence by the composition principles of AB grammar.

Relation to TDG

It is interesting to see the similarities between the unfolding indices introduced here and the "grammatical roles" in the framework of Debusmann and Duchier [7, 9]. Unfoldings can indeed be presented as dependency trees whose edges are labelled with unfolding indices (figure 4.4), a fact that enables the application of techniques developed for constraint parsing declarative dependency grammars to the parse problem for categorial grammars.

4.3 Stationary and transit constraints

The formalisation presented so far does allow neither for structural rules to increase its generative power beyond context-freeness, nor for multimodality. To test its extensibility, a recent proposal for structural constraints on Categorial Type Logics by Erk and Kruijff [11] was integrated into it. This section briefly presents the proposal and discusses the additional concepts needed for its implementation.

Not many methods are available for the processing of Categorial Type Logics in the generality that was discussed in chapter 2. The proof search

algorithm for $\mathsf{NL}\Diamond_{\mathcal{R}}$ proposed Moot [27] follows a "generate and test" approach: Given a CTL grammar (a base logic plus a package of structural rules) and an input sequence $S = w_1, \ldots, w_n$ of words,

- generate the set of AB derivations over all permutations of S, and then
- test for each such derivation, if by the structural rules of the grammar it can be rewritten into a new derivation with the axioms in the order actually encountered in S.

Erk and Kruijff [11] suggest two kinds of constraints to reduce the complexity of the search problem involved in the proof search algorithm. Both are obtained by an abstract interpretation of the rule set of the grammar: *Stationary constraints* impose restrictions on the generation phase by "locking" parts of the tree that cannot possibly be restructured. *Transit constraints* restrict the set of structural rules that can be applied to a given part of a generated tree, thereby pruning the search space for the testing phase of the algorithm.

Stationary constraints

Stationary constraints are derived from *stationary terms* for structural modes. A stationary term for a mode μ describes the effects that the application of a structural rule R has on tree material below nodes labelled with μ. The associativity rule

$$A \circ_{\mathsf{A}} (B \circ_{\mathsf{B}} C) \rightarrow (A \circ_{\mathsf{A}} B) \circ_{\mathsf{B}} C \tag{4.15}$$

for example does not affect the material in the left subtree of a node labelled with mode A (represented by the variable A), but removes material (C) from its right subtree: With respect to the rule, A is *left stationary*, but not *right-stationary*. If a rule only reorders material below a node labelled with a mode μ, but does not insert or remove any material, μ is called *stationary* with respect to that rule. The following definitions will formalise these concepts.

Definition 4.5 (ID map) Let $R = I \to O$ be a linear structural rule such that each node in I and O is labelled with a different mode or variable, and let D_I and D_O be the tree domains associated to I and O. Then the *ID map for* R, $\textsc{idmap}_R \in D_I \to D_O$, is the unique mapping that associates a node position $i \in D_i$ with a node position $o \in D_o$ if and only if i and o carry the same label.

It will often be convenient to identify nodes of a tree and their positions in the corresponding tree domain, and accordingly to treat $\textsc{idmap}_R$ as a mapping between nodes of different trees.

The next two definitions are due to Erk and Kruijff [11]. Given a node u of a tree τ, we write $\mathbb{L}_\tau(u)$ and $\mathbb{R}_\tau(u)$ for the subtrees rooted at u's left and right child, and $\mathbb{S}_\tau(u)$ for the subtree rooted at u.

Definition 4.6 (Stationary terms) Let R be a structural rule, and let $\textsc{idmap}_R$ be its ID mapping. If v_i is a node of I labelled by a structural mode μ, and $v_o = \textsc{idmap}_{R_i}(v_i)$, then $\mu(c_L, c_R)$ is a *stationary term* for R_i, where

- $c_L = \Downarrow$ ($c_R = \Downarrow$) if and only if exactly the variables that occur in $\mathbb{L}_I(v_i)$ ($\mathbb{R}_I(v_i)$) also occur in $\mathbb{L}_O(v_o)$ ($\mathbb{R}_O(v_o)$),
- otherwise $c_L = \Leftrightarrow$ ($c_R = \Leftrightarrow$) if and only if exactly those variables that occur in $\mathbb{S}_I(v_i)$ still occur in $\mathbb{S}_O(v_o)$, and
- $c_L = \Uparrow$ ($c_R = \Uparrow$) otherwise.

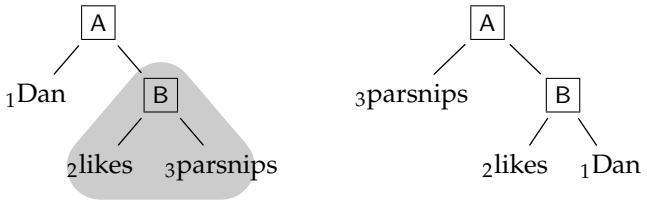
Definition 4.7 (Stationary modes) Let $\mathcal{R}$ be a set of structural rules and let $\Sigma_\mathcal{R}(\mu)$ be the set of stationary terms for rules in $\mathcal{R}$ with root μ. Then μ is

- *stationary* (for $\mathcal{R}$) if and only if for all $t \in \Sigma_\mathcal{R}(\mu)$, no child is $\Uparrow$,
- *left-stationary*, if and only if for all $t \in \Sigma_\mathcal{R}(\mu)$, the left child is $\Downarrow$,
- *right-stationary*, if and only if for all $t \in \Sigma_\mathcal{R}(\mu)$, the right child is $\Downarrow$.

The effect of the stationary constraints is the elimination of impossible tree configurations during the generation phase of the proof search algorithm. To illustrate this issue, consider the sentence

(1) $_1$Dan $_2$likes $_3$parsnips,

where the words are assigned the same categories as in figure 2.1. Moot's proof search algorithm generates two AB derivations for this input sequence (boxed letters mark modes of slash operators used in a derivation step):



The left derivation preserves the order of the words in the input sequence, the right derivation is yielded by a permutation of this order.

Assume now that mode A is right-stationary. This means, that all structural rules will apply either to material within the right subtree of the tree labelled with A in the left derivation (shaded), or will move this subtree as a whole, but will not remove any nodes from or add any nodes to it. In the right derivation, this condition is violated, as the node $_3$*parsnips* has been removed and replaced by the node $_1$*Dan*. Therefore, this derivation does not need to be considered as a candidate for rewriting.

Transit constraints

Transit constraints impose restrictions on the second phase of the proof search algorithm, the rewriting of the tree structures from the generation phase. They are derived from the *transit relation* of the grammar fragment in question, which describes how structural rules can affect tree fragments identified by *transit patterns*.

Definition 4.8 (Transit pattern) Let $\mathcal{M}$ be the set of modes for a given grammar fragment, and let trn be a special label not occuring in $\mathcal{M}$, called the *transit label*. A *transit pattern* then is a triple $\langle p, d, c\rangle$ where

- p and c are labels from the set $\mathcal{M} \uplus \{\mathsf{trn}\}$, called the *parent* and the *child* label, such that exactly one of them is the transit label, and
- $d \in \{\lhd, \rhd\}$ is a *direction*.

Transit patterns will usually be written $(\mathsf{trn})\mu$, $(\mu)\mathsf{trn}$ (direction $\lhd$) and $\mu(\mathsf{trn})$, $\mathsf{trn}(\mu)$ (direction $\rhd$), where μ is a mode label, and the child label is written in parentheses.

Definition 4.9 (Occurs at) A transit pattern $\langle p, \lhd, c\rangle$ ($\langle p, \rhd, c\rangle$) *occurs* in a tree *at the node* u if

- $p = \mathsf{trn}$, and u has a left (right) child u' labelled with c, or
- $c = \mathsf{trn}$, and u is the left (right) child of a node u' labelled with p.

In both cases, the node u is called *transit node*, the node u' *context node* of the occuring pattern.

To get an understanding of how the effects of structural rules are encoded in the transit relation, consider again the associativity rule in 4.15:

$$A \circ_{\mathsf{A}} (B \circ_{\mathsf{B}} C) \to (A \circ_{\mathsf{A}} B) \circ_{\mathsf{B}} C \tag{4.15}$$

On its left hand side, the transit pattern $\mathsf{A}(\mathsf{trn})$ occurs at the node $\circ_{\mathsf{B}}$, on its right hand side, $(\mathsf{A})\mathsf{trn}$; in the transit relation, there would be an edge between the two patterns.

To obtain the transit constraints, each transit pattern is annotated with the labels of the nodes at which the pattern occurs in a tree generated during the first phase of the proof search algorithm. The structural rules then license

the "travelling" of these labels to other transit patterns along the edges of the transit relation. The thus annotated transit relation is called a *logical roadmap*. Trees resulting from the rewriting phase have to obey the constraint that each transit pattern can only occur for labels that are licensed by a logical roadmap. For a formalisation of these concepts, the reader is referred to the original paper by Erk and Kruijff [11].

Transit constraints as abstract interpretation

The analysis of structural rules in terms of the effects that they have on parts of trees identified by transit patterns is a typical example of abstract interpretation. Similar to the interval abstraction presented in section 3.3, it allows to draw conclusions potentially pruning the search space, without the danger of losing solutions to the overall problem.

It is obvious that the abstraction from structural rules to the transit relation is not complete, as a structural rule in the general case modifies several transit patterns at the same time. The consequence of the abstraction's incompleteness is that the constraints proposed by Erk and Kruijff license *all* valid derivations, but not *only* valid derivations: The outcome derivations of the constrained proof search algorithm are a superset of the derivations actually licensed by the grammar.

Implementing stationary constraints

To give an example for how the framework presented in section 4.2 can be extended by constraints like the ones proposed by Erk and Kruijff, the formalisation of stationary constraints will be demonstrated. The formalisation of transit constraints [11] does not require any additional concepts.

The crucial additional concept needed to capture the effects of stationary constraints in the unfolding encoding is the notion of the *younger sisters* of an unfolding index and its associated subcat item.

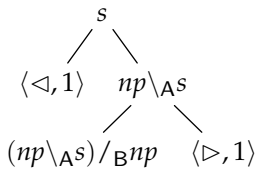


Figure 4.5: Example for definition 4.11

Apart from the "horizontal" dimension of the unfolding that is encoded in the grid, there also is a "vertical" dimension: In the unfolding given in figure 4.3(c), the hole with the index $\langle \triangleright, 1 \rangle$ lies *below* the hole with the index $\langle \triangleleft, 1 \rangle$. With reference to a flattened tree like the one in figure 4.4, the left subcat *np* will therefore be referred to as a *younger sister* of the right subcat *np*. The definition of this concept is straightforward:

Definition 4.10 (Younger sister) Let U be an unfolding with two holes H_1 and H_2, and let $\mathfrak{I}_1$ and $\mathfrak{I}_2$ be the indices of these holes. Then $\mathfrak{I}_2$ is called a *younger sister* of $\mathfrak{I}_1$ if the path from the root of the unfolding to H_2 is longer than the path from the root to H_1.

Another relevant piece of information is the mode under which an unfolding index/subcat item is linked to the anchor of its unfolding.

Definition 4.11 (Mode associated to an unfolding index) Let U be an unfolding, and let $\mathfrak{I}$ be an unfolding index in U. The *mode associated to* $\mathfrak{I}$ is the mode of the slash operator by which the category of the hole at $\mathfrak{I}$ was linked in U.

For example, in the unfolding shown in figure 4.5, the associated mode for the unfolding index $\langle \triangleleft, 1 \rangle$ is A, while the associated mode for $\langle \triangleright, 1 \rangle$ is B.

Assume that the input for the proof search algorithm is a sequence $w_1, \ldots, w_n$ of words. Recall that in the present framework, every word corresponds to exactly one unfolding. Therefore, each subtree of the derivation can be mapped

to a set of indices identifying those unfoldings/words that this subtree has been composed of. This set will be called the *yield* of the corresponding subtree. To say that a mode that labels a node u is stationary then amounts to stating that the union of the yields of the daughters of u is convex – the composing unfoldings can swap place, but no material can move out. Similarly, if a mode labelling a node u is left-stationary (right-stationary), then the yield of the subtree rooted at the left (right) daughter of u must be convex – the unfoldings that it has been composed of must be adjacent.

To formalise these ideas in the unfolding framework, the tree configuration problem is amended by three new functions:

$$\begin{aligned} \textsc{yield} &\in \mathit{Nodes} \to \mathfrak{P}(\mathbb{N}) \\ \textsc{gridYields} &\in \mathit{Nodes} \to \mathcal{G}(\mathcal{C}) \to \mathfrak{P}(\mathbb{N}) \\ \textsc{youngerSisterYields} &\in \mathit{Nodes} \to \mathcal{G}(\mathcal{C}) \to \mathfrak{P}(\mathbb{N}) \end{aligned}$$

The yield $\textsc{yield}(N)$ of a node N is a set of integers, giving the positions of nodes weakly below N. The gridYields functions gives the yield of the the daughters of a node at a given position in the grid:

$$\forall N \in \mathit{Nodes}\colon \forall \mathfrak{I} \in \mathcal{G}(\mathcal{C})\colon \textsc{gridYields}(N)(\mathfrak{I}) = \{\, \textsc{yield}(M) \mid M \in \textsc{subcat}(N)(\mathfrak{I}) \,\}\,. \quad (4.16)$$

youngerSisterYields maps a node's grid position to the yield of the younger sisters (youngerSisters) at that position:

$$\forall N \in \mathit{Nodes}\colon \forall \mathfrak{I} \in \mathcal{G}(\mathcal{C})\colon \textsc{youngerSisterYields}(N)(\mathfrak{I}) = \{\, \textsc{yield}(M_1) \mid M_1 \in \{\, \textsc{subcat}(M_2)(\mathfrak{I}) \mid M_2 \in \textsc{youngerSisters}(N)(\mathfrak{I}) \,\} \,\}\,. \quad (4.17)$$

If a subcat item is attached to an anchor under a mode from the set *Stationaries* of all stationary modes, then the yield of its associated node must be a convex set of position indices. The function MODES gives the modes associated to a given set of nodes. The convexity constraint is realised by a propagator convex.

$$\forall N \in \mathit{Nodes}\colon \forall \mathfrak{I} \in \mathcal{G}(\mathcal{C})\colon |\mathit{Stationaries} \cap \textsc{modes}(\textsc{subcat}(N)(\mathfrak{I}))| = 0 \quad \vee \quad \mathsf{convex}(\textsc{youngerSisterYields}(N)(\mathfrak{I})) \tag{4.18}$$

If a subcat item is attached to an anchor under a mode that is left- or right-stationary (specified by the sets $\mathit{FieldStationaries}(\triangleleft)$ and $\mathit{FieldStationaries}(\triangleright)$), then that part of the yield of its associated node that lies in the respective field must be a convex set of positions.

$$\forall N \in \mathit{Nodes}\colon \forall \mathfrak{I} \in \mathcal{G}(\mathcal{C})\colon \forall F \in \{\triangleleft, \triangleright\}\colon |\mathit{FieldStationaries}(F) \cap \textsc{modes}(\textsc{subcat}(N)(\mathfrak{I}))| = 0 \quad \vee \quad \Xi(N, \mathfrak{I}, F) \tag{4.19}$$

where

$$\Xi(N, \mathfrak{I}, F) = \mathsf{convex}(\textsc{youngerSisterYields}(N)(\mathfrak{I}) \cap \textsc{gridYields}(N)(\mathfrak{I}))$$

if $\mathfrak{I}$ is in field $F \in \{\triangleleft, \triangleright\}$, and $\Xi(N, \mathfrak{I}, F) = \text{true}$ otherwise.

Summary

Together with the restrictions formulated in section 4.2, the new constraints 4.16–4.19 constrain the generation phase of the proof search algorithm. Each generated tree is the starting point for the application of structural rules. This rewriting process is constrained by the stationary constraints and the transit constraints and eventually yields a set of destination trees, corresponding to potential derivations of the original sentence [11].

Chapter 5

The Implementation

One of the central objectives of this project was the implementation of the framework presented in the previous section, and the stationary and transit constraints proposed by Erk and Kruijff. This implementation has been carried out using the Mozart/Oz programming system [28].

5.1 General architecture

The architecture of the implementation (figure 5.1) reflects the two-step strategy developed in section 4.3:

1. In the first step, a grammar definition, given as an XML document, is read in and analysed to obtain the stationary constraints and the transit relation. The stationary constraints are used in a constraint solver for the *starting tree* problem: Generate those AB derivations trees over permutations of a given input sequence of words that are licensed by stationary modes.

2. In the second step, the user can select a specific starting tree to obtain the corresponding logical roadmap, which induces the transit constraints. These constraints are then used in a second constraint solver to compute the *destination trees* – those derivations that can potentially be obtained from the starting tree by applying the structural rules to it.

The different components of the implementation can perhaps best be introduced by a concrete example.

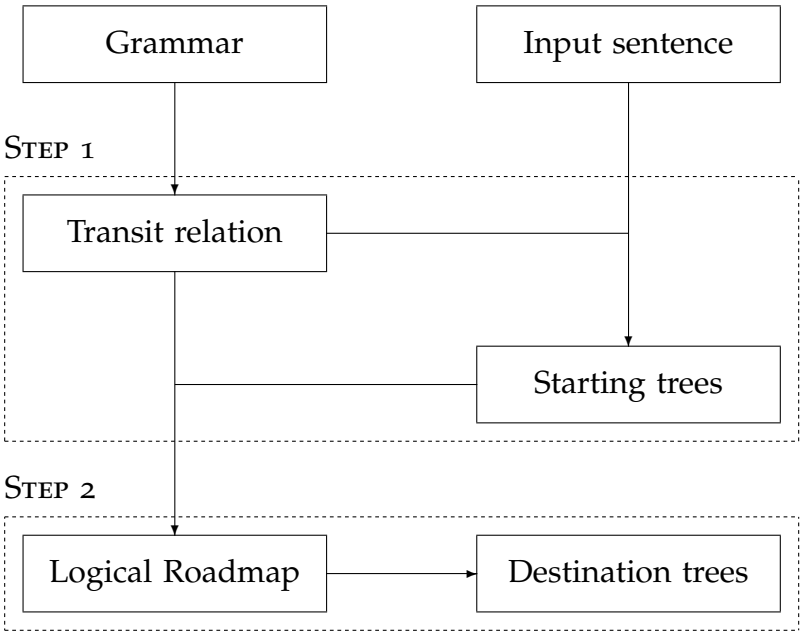


Figure 5.1: Architecture of the implementation

5.2 Example

Consider the grammar fragment given in table 5.1, which is also used by Erk and Kruijff [11]. The categories and structural rules of this fragment license a number of sentences that illustrate an instance of word-order variation in German:

(1) ..., dass Maria verspricht einen Roman zu schreiben.

(2) ..., dass Maria einen Roman verspricht zu schreiben.

(3) ..., dass einen Roman zu schreiben Maria verspricht.

Assume that the system is initialised with the grammar fragment and sentence (2). The initialisation starts the Oz Explorer [31], an interactive tool that will display the search tree for the starting tree problem. The user now has the choice between several "information actions" (figure 5.2) to inspect the solutions of the problem:

$$A \circ_{\mathsf{con}} (B \circ_{\mathsf{dc}} C) \rightarrow B \circ_{\mathsf{dc}} (A \circ_{\mathsf{con}} C) \quad \text{(R1)}$$

$$A \circ_{\mathsf{sc}} (B \circ_{\mathsf{con}} C) \rightarrow (A \circ_{\mathsf{sc}} B) \circ_{\mathsf{con}} C \quad \text{(R2)}$$

$$A \circ_{\mathsf{con}} (B \circ_{\mathsf{dc}} C) \rightarrow (B \circ_{\mathsf{dc}} C) \circ_{\mathsf{con}} A \quad \text{(R3)}$$

dass :	$relc/_{\mathsf{rel}}scon$
Maria :	np
verspricht :	$(np\backslash_{\mathsf{sc}}scon)/_{\mathsf{con}}(np\backslash_{\mathsf{sc}}zuinf)$
einen :	$np/_{\mathsf{d}}np$
Roman :	np
zu_schreiben :	$np\backslash_{\mathsf{dc}}(np\backslash_{\mathsf{sc}}zuinf)$

Table 5.1: Sample grammar fragment [11]

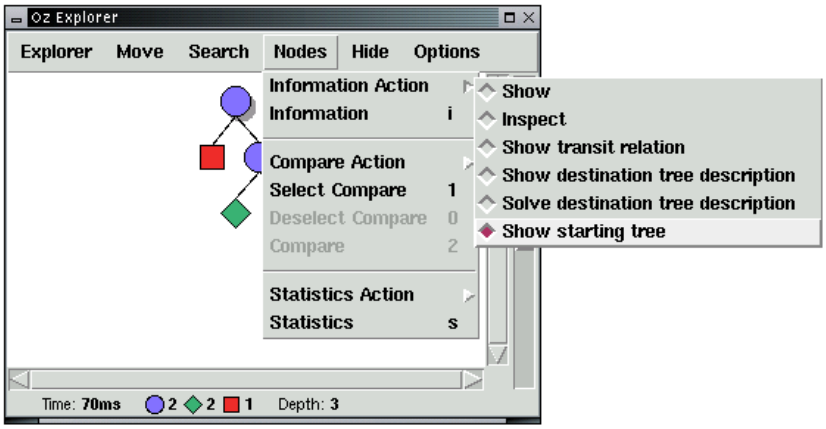


Figure 5.2: The Information Action menu

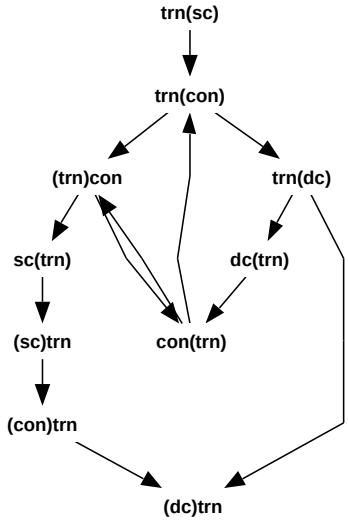


Figure 5.3: The transit relation for the grammar fragment in table 5.1

Show transit relation This will show a graphical representation of the transit relation induced by the grammar, as rendered by the DaVinci graph drawing software [3]. The transit graph for the example grammar fragment is shown in figure 5.3.

Show starting tree This information action will show a tree representation of the selected solution, using Guido Tack's tree drawing widget [35]. The example grammar fragment licenses two starting trees, which are given in figure 5.4.

Show destination tree description The selected starting tree is used to annotate the transit relation with the labels of the nodes that the transit patterns occur at, eventually resulting in a logical roadmap that induces the transit constraints. Transit constraints and stationary constraints yield an underspecified description of the destination trees. A graphical representation of this description will be shown in the DaVinci window.

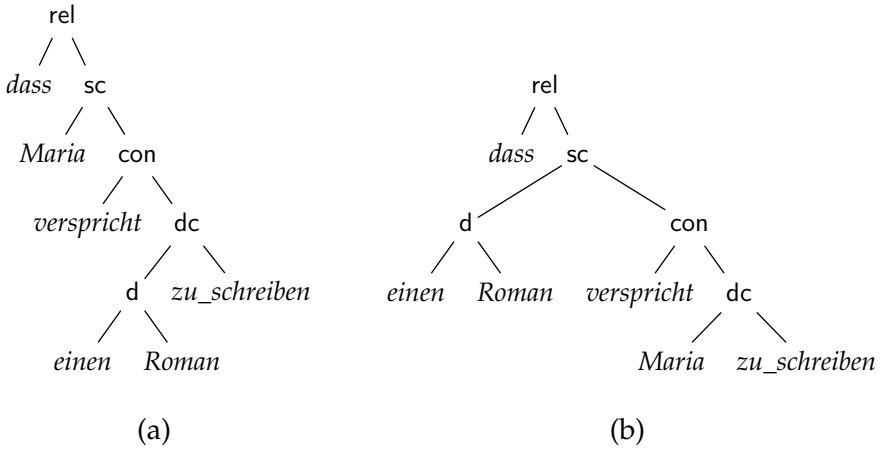


Figure 5.4: Starting trees for the grammar fragment in table 5.1

The destination tree description for the starting tree in figure 5.4(a) is shown in figure 5.5. The solid lines depict *dominance constraints*: The label at the end of the arrow must be attached to a node occuring strictly below the node that carries the label at the source of the arrow. Dominance constraints are obtained from the stationary constraints and the transit constraints [11]. The dashed lines correspond to *precedence constraints*, which restrict the linear order of the leaves. These constraints are obtained from the order of the words in the input sequence.

Solve destination tree description The destination tree description of the selected starting will be sent to and solved by a new instance of the Oz Explorer. For the starting tree in figure 5.4(a), there is one destination tree, which is shown in figure 5.6. It can be obtained from its starting tree by a single application of the structural rule R1.

⋆ ⋆ ⋆

The implementation has been released under the terms and conditions of the GNU Public License and made publicly available through the Mozart Global Users' Library (MOGUL) under the ID `mogul:/kuhlmann/concat`.

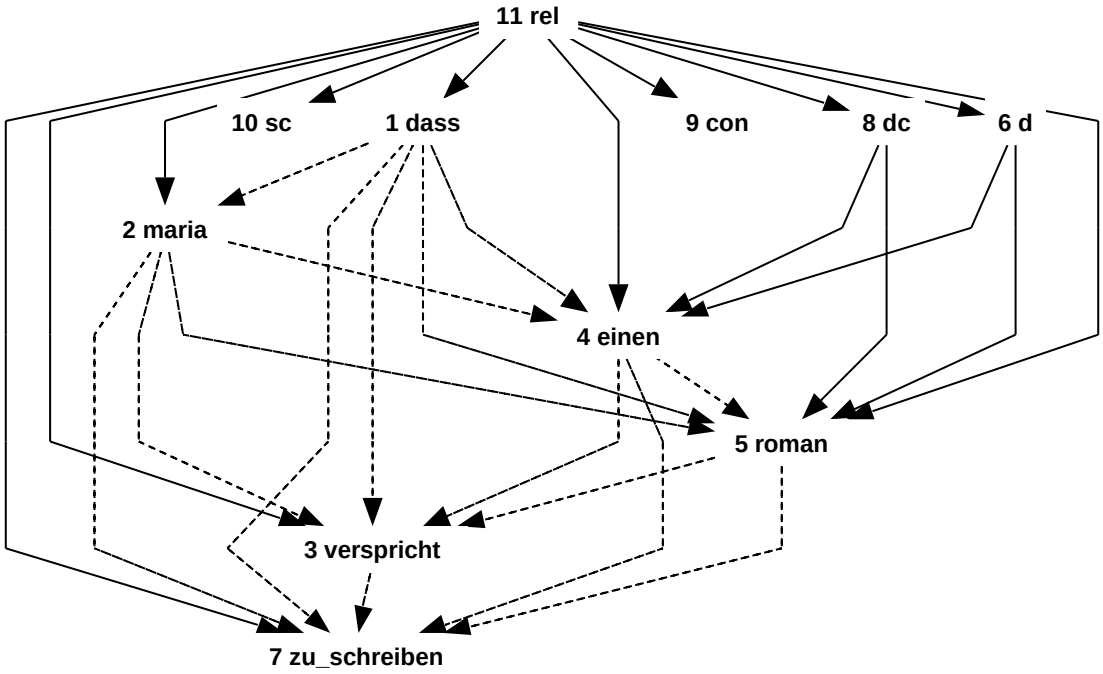


Figure 5.5: Destination tree description for the starting tree in figure 5.4(a)

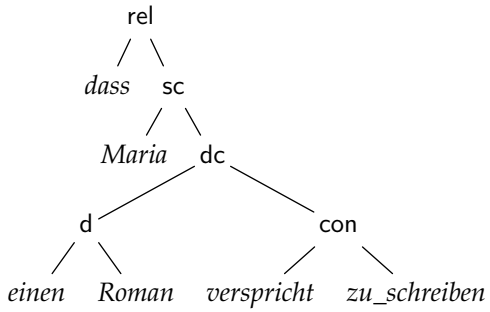


Figure 5.6: Destination tree for the starting tree in figure 5.4(a)

Chapter 6

Conclusions and Outlook

This thesis has shown how constraint programming can be applied to the processing of Categorial Type Logics (CTLs). It has presented a novel formalisation of the parsing task for categorial grammars as a tree configuration problem (section 4.2), and demonstrated the extensibility of this framework by integrating into it a recent proposal for constraints on structural control (section 4.3). The constraint framework is a promising foundation for detailed studies on the real-world complexity of CTLs and the development of new grammar fragments. One of its main advantages is its fundamentally declarative nature, which allows a close correspondence between formalisation and implementation. This property has been demonstrated by the prototype system implemented during this project.

Several questions remain to be answered in order to make the present framework a fully-fledged parser for CTLs.

Validation First, due to the incompleteness of the structural constraints discussed in section 4.3, the prototype system still produces a *superset* of the legitimate derivations. To allow the comparison with other systems, a *validation* component would be needed, that checks derivations for their validity. The design and implementation of this component poses a challenge, as validation can easily become a "generate and test" step in disguise [11]. To avoid this, one would probably need to make efficient usage of the information accumulated in the logical roadmap. However, the details of the validation step require further investigation.

Coverage The time frame for this project did not allow the evaluation of the proposed method on larger grammar fragments. To acquire such fragments, it would be interesting, for example, to combine the idea of using CTLs to simulate Combinatory Categorial Grammar (CCG) [20] with the recent work of Hockenmaier and Steedman on the automated extraction of CCG lexicons from large text corpora [16]. The evaluation of the constraint parsing framework on such real-world data would hopefully provide us with a better understanding of the *practical* efficiency of this approach.

Unary modalities An important extension of the framework proposed here is the treatment of unary modalities. There are two ways in which they could be integrated into the present formalisation: They could either be modelled by additional nodes in the parse tree, or encoded at the existing nodes as "structural features". The former alternative would compromise the non-redundant encoding and decrease the power of constraint propagation, and is therefore undesirable. The features approach would have to take into account that structures can be "locked" with several modes, and that the order of these modes in general will matter for the applicability of structural rules. With respect to the unfolding encoding, this would require a more fine-grained notion of when two unfoldings can plug together.

Hypothetical reasoning The parsing framework presented here does not currently allow hypothetical reasoning (cf. section 2.4). Furthermore, considering that one of the crucial features of the unfolding encoding is the bijection between words in the input sentence and nodes in the parse tree, it is not at all obvious how it could provide for hypotheticals in the first place. One possibility would be to adapt compilation techniques [15] to the present framework. Another interesting direction would be to explore the connections between the unfolding encoding and partial proof trees [17], in which hypothetical reasoning is possible through the stretching operation.

PARALLELISM A final proposal for further work concerns the general architecture of the processing itself: The current model is linear; it first generates the starting trees and then for each of these starting trees produces the destination trees. It would certainly be interesting to investigate if they could be executed in *parallel*, and if such an architecture could set free synergy effects, such that the two processes would mutually constrain each other. A similar architecture is used in the parser for Topological Dependency Grammar (TDG) [7], and has proven to be very successful.

Bibliography

[1] Kazimierz Ajdukiewicz. Die syntaktische Konnexität. *Studia Philosophica*, 1:1–27, 1935.

[2] Krzysztof R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1–2):179–210, 1999.

[3] b-novative GmbH, Bremen. DaVinci Presenter: Automatische Visualisierung von Strukturen und Netzen. http://www.davinci-presenter.de/.

[4] Roman Barták. Constraint programming: in pursuit of the Holy Grail. In *Proceedings of WDS'99 (invited lecture)*, 1999. URL http://kti.ms.mff.cuni.cz/~bartak/constraints/.

[5] Bob Carpenter. The Turing completeness of multimodal categorial grammars. In Jelle Gerbrandy, Maarten Marx, Maarten de Rijke, and Yde Venema, editors, *JFAK. Essays Dedicated to Johan van Benthem on the Occasion of his 50th Birthday*. Amsterdam University Press, 1999.

[6] Patrick Cousot. Abstract interpretation based formal methods and future challenges, invited paper. In Reinhard Wilhelm, editor, *Informatics – 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer, 2001.

[7] Ralph Debusmann. A declarative grammar formalism for dependency grammar. Master's thesis, Saarland University, Saarbrücken, 2001.

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

[9] Denys Duchier. Configuration of labeled trees under lexicalized constraints and principles. To appear in the Journal of Language and Computation, December 2000.

[10] 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.

[11] Katrin Erk and Geert-Jan M. Kruijff. A constraint-programming approach to parsing with resource-sensitive categorial grammar. In *Proceedings of the 7th International Workshop on Natural Language Understanding and Logic Programming (NLULP'02)*, Copenhagen, 2002.

[12] Herman Hendriks. *Studied flexibility. Categories and Types in Syntax and Semantics.* PhD thesis, ILLC, University of Amsterdam, Amsterdam, 1993.

[13] Mark Hepple. *The Grammar and Processing of Order and Dependency: a Categorial Approach*. PhD thesis, University of Edinburgh, Edinburgh, 1990.

[14] Mark Hepple. Mixing modes of linguistic description in categorial grammar. In *Proceedings of the Seventh Conference of the European Chapter of the Association for Computational Linguistics (EACL–7)*, pages 127–132, Dublin, March 1995.

[15] Mark Hepple. An Earley-style predictive chart parsing method for Lambek grammars. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics (ACL'99)*, pages 465–472, Maryland, June 1999.

[16] Julia Hockenmaier and Mark Steedman. Acquiring compact lexicalized grammars from a cleaner treebank. In *Proceedings of the Third International Conference on Language Resources and Evaluation*, 2002.

[17] Arivind K. Joshi and Seth Kulick. Partial proof trees as building blocks for a categorial grammar. *Linguistics and Philosophy*, 20(6):637–667, 1997.

[18] Esther König. Parsing as natural deduction. In *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics*, pages 272–279, Vancouver, BC, June 1989.

[19] Donald E. Knuth. Generating all permutations, 2002. Draft of section 7.2.1.2 of *The Art of Computer Programming*, http://www-cs-faculty.stanford.edu/~knuth/fasc2b.ps.gz.

[20] Geert-Jan Kruijff and Jason Baldridge. Relating categorial type logics and CCG through simulation. URL http://www.iccs.informatics.ed.ac.uk/~jmb/simulation.ps.gz. Draft, April 2000.

[21] Joachim Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, 65:154–170, 1958.

[22] Stanislaw Lesniewski. Grundzüge eines neuen Systems der Grundlagen der Mathematik. *Fundamenta Mathematicae*, 14, 1929.

[23] Kim Marriott and Peter J. Stuckey. *Programming with Constraints*. MIT Press, Cambridge, MA, 1998.

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

[25] Michael Moortgat. Categorial type logics. In Johan van Benthem and Alice ter Meulen, editors, *Handbook of Logic and Language*, pages 93–177. The MIT Press, Cambridge, MA, 1997.

[26] Richard Moot. Grail: A tool for the development and prototyping of grammar fragments for categorial logics. http://www.let.uu.nl/~Richard.Moot/personal/grail.html.

[27] Richard Moot. *Proof Nets for Linguistic Analysis*. PhD thesis, Utrecht University, Amsterdam, 2002.

[28] Mozart Consortium. Mozart/Oz. http://www.mozart-oz.org/.

[29] Mati Pentus. Lambek grammars are context free. In *Proceedings of 8th Annual IEEE Symposium on Logic in Computer Science, LICS'93, Montreal, Canada, 19–23 June 1993*, pages 429–433. IEEE Computer Society Press, Los Alamitos, CA, 1993.

[30] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, NJ, 1995.

[31] Christian Schulte. *Programming Constraint Services*, volume 2302 of *Lecture Notes in Artficial Intelligence*. Springer-Verlag, Berlin, Germany, 2002.

[32] Christian Schulte and Peter J. Stuckey. When do bounds and domain propagation lead to the same search space. In Harald Søndergaard, editor, *Third International Conference on Principles and Practice of Declarative Programming*, Florence, September 2001. ACM Press.

[33] Gert Smolka. The Oz Programming Model. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer, Berlin, 1995.

[34] Mark Steedman. *The Syntactic Process*. The MIT Press, 2000.

[35] Guido Tack. TkTreeWidget, a tree drawing widget for Tk. http://www.mozart-oz.org/mogul/info/tack/TkTreeWidget.html, 2002.

[36] Mary McGee Wood. *Categorial Grammars*. Linguistic Theory Guides. Routledge, London, 1993.