Thesis for the Degree of Doctor of Philosophy

Expressivity and Complexity of the Grammatical Framework

Peter Ljunglöf

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE–412 96 Göteborg, Sweden

Göteborg, November 2004

Expressivity and Complexity of the Grammatical Framework
Peter Ljunglöf
ISBN 91-628-6331-2

Technical Report no. 31D
ISSN 1651-4971
School of Computer Science and Engineering

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE–412 96 Göteborg, Sweden
Telephone +46 (0) 31–772 1000

Printed at Chalmers, Göteborg, Sweden, 2004

Varje varelse, varje skapelse, varje dröm som människan någonsin drömt finns här. Ni formade dem i era drömmar och fabler och i era böcker, ni gav dem form och substans och ni trodde på dem och gav dem makt att göra det och det ända tills de fick eget liv. Och sedan övergav ni dem.

Lundwall (1974, p. 114)

Abstract

This thesis investigates the expressive power and parsing complexity of the GRAMMATICAL FRAMEWORK (GF), a formalism originally designed for displaying formal propositions and proofs in natural language. This is done by relating GF with two more well-known grammar formalisms; GENERALIZED CONTEXT-FREE GRAMMAR (GCFG), best seen as a framework for describing various grammar formalisms; and PARALLEL MULTIPLE CONTEXT-FREE GRAMMAR (PMCFG), an instance of GCFG.

Since GF is a fairly new theory, some questions about expressivity and parsing complexity have until now not been answered; and these questions are the main focus of this thesis. The main result is that the important subclass *context-free* GF is equivalent to PMCFG, which has polynomial parsing complexity, and whose expressive power is fairly well known.

Furthermore, we give a number of tabular parsing algorithms for PMCFG with polynomial complexity, by extending existing algorithms for context-free grammars. We suggest three possible extensions of GF/PMCFG, and discuss how the expressive power and parsing complexity are influenced. Finally, we discuss the parsing problem for unrestricted GF grammars, which is undecidable in general. We nevertheless describe a procedure for parsing grammars containing higher-order functions and dependent types.

Keywords: *Grammatical Framework, generalized context-free grammar, multiple context-free grammar, context-free rewriting systems, type theory, expressive power, abstract syntax, linearization, parsing*

Acknowledgments

Although I am the sole writer of the text in this thesis, it would never have been written without the support of a number of people.

I vaguely recall from when I started my five year PhD studies, that my supervisor Aarne Ranta had an idea of investigating the expressive power and parsing complexity of the GRAMMATICAL FRAMEWORK; and after three years, one daughter and a Licenciate thesis, the idea had finally reached my brain. Now, after two more years, one son, and some reinvention of the wheel, the idea together with some results of this investigation is available in printed form. Apart from planting the seed to this thesis, Aarne has also patiently listened to, answered, explained, rejected and sometimes accepted my questions and ideas; always taking his time, even when there has been none.

Some of the reasons why I began my PhD studies in the first place are Robin Cooper and Jan Smith, who made me interested of doing research. Together with people like Bengt Nordström, Jan Smith, Torbjörn Lager, Joakim Nivre, Koen Claessen, Josef Svenningsson and Paul Callaghan, they have throughout the years taken their time for discussions about type theory, grammar formalisms and parsing algorithms; the three research areas that form the foundational origins of this thesis.

For very helpful and insightful comments on, and proof-reading of, earlier versions of this manuscript, I am grateful to Aarne Ranta, Joakim Nivre, Bernard Lang, Robin Cooper and Kristofer Johannisson.

I am honoured to have Bernard Lang as my opponent; some parts in this thesis would be much more difficult to finalize without some of his publications.

During the final year of writing this thesis, my research has been enabled by support from the EU project TALK.[1]

I'm sometimes aware that there is a world outside the actual thesis writing; even though it has been far away the last year or so. From this world I would like to thank my current and previous office mates, and the other fellow current and previous PhD students, for all the social events I almost certainly fail to attend. The people of the departments of Computing Science and Linguistics have succeeded in creating a research-friendly atmosphere; and my students have been very good in creating a teaching-friendly atmosphere.

There are several people, both inside the research community and outside in the real world, I would like to thank for interesting discussions on whatever topic, or simply just for being around; but this list would most certainly be both long and incomplete, so I leave its completion as an exercise to the interested reader.

I would like to thank all my friends and relatives who never will understand what I do for a living, but still accept me as the person I happen to be.

Finally, my greatest thanks go to my beloved family; Saga, Signe, Elis and Svea, who continue to drag my attention away from research, reminding me that there really is a world outside of this department.

Göteborg, November 2004

[1]TALK (Talk and Look, Tools for Ambient Linguistic Knowledge), IST-507802

Contents

Figures

Notations

This is a list of the notations that are used in this thesis. Note that a symbol/notation can occur in several places in this list if it is used with different meanings.

$\mathbb{N}$	the set of natural numbers
$\mathbb{N}_n$	the finite set $\{ 0, \ldots, n-1 \}$
i, j, k, n, m	natural numbers
G	a grammar
Σ	a set of terminals
$\mathcal{C}$	a set of categories
A, B	categories and types
C	a basic category
S	the starting category of a grammar
$\mathcal{F}$	a set of function symbols
f, g	function symbols
a, b	constants (functions without arguments)

$\mathcal{R}$	a set of rules
R	a grammar rule
δ	an arity (the number of arguments of a rule)
$\mathcal{T}$	a set of trees
t	an abstract term; a tree
$\mathcal{C}$	a chart
θ	a chart item
$\mathcal{L}$	a language
s	a string
w	the input string
T	a linearization type
ϕ, ψ	linearizations
α, β, γ	sequences in linearization rules
P	a parameter type
p	a parameter; a parameter pattern
π	a parameter record
r, s	record labels
σ	a path (sequence of labels and parameters)
Σ	a set of labels or paths
ρ	a range
Γ	a record (or a general datastructure) of ranges
σ	a substitution
π	a projection
θ	a permutation

Abbreviations

In this thesis we refer to many different grammar formalisms, and use abbreviations whenever possible; the most common are listed below.

GF	Grammatical Framework
cf-GF	Context-free GF
CFG	Context-free grammar
GCFG	Generalized CFG
MCFG	Multiple CFG
PMCFG	Parallel MCFG
LMCFG	Linear MCFG
LCFRS	Linear context-free rewriting system
LMG	Literal movement grammar
s-LMG	Simple LMG
RCG	Range concatenation grammar
poms-CFG	Partially ordered multiset CFG
HG	Head grammar
TAG	Tree adjoining grammar
(L)IG	(Linear) indexed grammar
(C)CG	(Combinatory) categorial grammar
HPSG	Head-driven phrase structure grammar
LFG	Lexical functional grammar

Chapter 1

Introduction

This thesis investigates the expressive power and parsing complexity of the *Grammatical Framework* (GF; Ranta, 2004a), a formalism originally designed for displaying formal propositions and proofs in natural language. This is done by relating GF with two more well-known grammar formalisms; *generalized context-free grammar* (GCFG; Pollard, 1984), best seen as a framework for describing various grammar formalisms; and *parallel multiple context-free grammar* (PMCFG; Seki et al., 1991), an instance of GCFG.

This first chapter introduces the problem setting, and discusses some questions about expressivity and parsing complexity which have until now not been answered, and are the main focus of the thesis. The chapter also contains introductions to the areas of expressivity and parsing complexity, and to the grammar formalisms GF and GCFG. Finally, an overview of the thesis is given, in which the main results are discussed.

Since GF is a fairly new theory, some questions about expressivity and parsing complexity have until now not been answered; and these questions are the main focus of this thesis. The main result is that the important subclass *context-free* GF is equivalent to PMCFG, which has polynomial parsing complexity, and whose expressive power is fairly well known. Furthermore, a number of parsing algorithms for PMCFG are given; some possible extensions are studied; and the idea of using the algorithms when parsing unrestricted GF grammars is discussed.

1.1 Motivation for this thesis

This thesis investigates the relations between the following grammar formalisms, all sharing the idea of separating abstract and concrete syntax;

Grammatical framework (gf; Ranta, 2004a) is a formalism originally designed for displaying formal propositions and proofs in natural language, but has since then evolved into a formalism suited for describing both the semantics and the syntax of natural languages. The representation of the abstract syntax is intuitionistic type theory, and the concrete syntax is a restricted variant of a functional programming language. gf is a very general formalism, since the abstract syntax is a logical framework; it is e.g. possible to formulate an undecidable proposition as an undecidable parsing problem inside gf.

Generalized context-free grammar (gcfg; Pollard, 1984) is also a very general grammar formalism, originally designed to give a formal interpretation for head grammar. The abstract syntax is a context-free grammar, whereas the concrete syntax is only vaguely specified. Thus gcfg can be seen as a framework for describing various grammar formalisms.

Parallel multiple context-free grammar (pmcfg; Seki et al., 1991) is an instance of gcfg, where the concrete syntactical structures are tuples of strings. It is known that pmcfg parsing is polynomial in the length of the input string.

Since gf is a fairly new theory, some questions about expressivity and parsing complexity have until now not been answered, especially for a very important subclass called *context-free* gf (from now on written *cf*-gf);

(1) What is the expressive power of *cf*-gf; i.e. what language constructs can the formalism express?

(2) What is the parsing complexity of *cf*-gf; i.e. are there efficient parsing algorithms?

These two questions are the main focus of this thesis. The area of research can therefore be narrowed to formal language theory and parsing algorithms, two areas that have tight connections; if two formalisms are strongly equivalent, the one can be used to parse grammars from the other.

The main result in this thesis is that *cf*-gf and pmcfg are equivalent formalisms. Since pmcfg has a polynomial parsing algorithm and its expressive power is fairly well known, the result answers both question (1) and (2). As a side-effect, new parsing algorithms for gf can be developed using the simpler pmcfg formalism. As a further answer to question (2), a number of new parsing algorithms for pmcfg are developed, all being polynomial in the length of the input.

As mentioned above, the concrete syntax of gf is a restricted functional programming language. A natural question then arises;

(3) How can the concrete syntax be extended, e.g. by adding new operations?

(4) What happens to the expressive power and parsing complexity?

These two questions are partially answered by giving three possible extensions; intersection, disjunction and interleave. Two of these (intersection and disjunction) are strict extensions in the sense that the new formalism can express a wider range than previously. The parsing complexity is still polynomial for intersection; in fact, it is shown that with this extension, *cf*-GF and PMCFG describe exactly the class of languages recognizable in polynomial time.

Finally, the thesis addresses the parsing problem for full GF;

(5) Can the polynomial parsing algorithms for *cf*-GF be of use when parsing unrestricted GF grammars?

The question is partially answered for a subclass of GF, which is larger than *cf*-GF, but cannot handle all possibilities of a general logical framework.

1.2 Expressivity and parsing complexity

1.2.1 Expressive power

We use the standard definition of what constitutes a language. In short, a language is a set of strings, where we write e.g. $a^n b^n$ for the set $\{ a^n b^n \mid n \geq 0 \}$.

What context-free grammars cannot express

The following non-regular constructions can be expressed by context-free grammars (see e.g. Hopcroft and Ullman, 1979);

- Nesting, exemplified by the language $a^n b^n$;
- Reverse copying,[1] exemplified by the language $\{ w\, w^R \mid w \in (a \cup b)^* \}$.

From the *pumping lemma* for context-free languages (see e.g. Hopcroft and Ullman, 1979), it is possible to show that the following constructions are not possible to express with a context-free grammar;

- Multiple agreement, exemplified by the language $a^n b^n c^n$;
- Crossed agreement, exemplified by the language $a^n b^m c^n d^m$;

[1]By the operation w^R we mean the reverse of w.

- Duplication, exemplified by the language $\{ w\, w \mid w \in (a \cup b)^* \}$.

However, there is linguistic evidence (Joshi, 1985; Shieber, 1985) that these three constructions occur in natural languages. Partly for this reason, but mostly because it simplifies grammar writing, more expressible grammar formalisms have been suggested.

Mildly context-sensitive grammar formalisms

The next step after context-free grammars in the Chomsky hierarchy is *context-sensitive grammars* (Chomsky, 1959). Unfortunately, this step is quite big; context-sensitive grammars can express an unnecessary large class of languages, with the drawback that parsing is no longer polynomial in the length of the input. Joshi (1985) suggested therefore the notion of *mild context-sensitivity* to capture the formal power needed for defining natural languages. A grammar formalism is mildly context-sensitive if it has the following four properties;

- It can express any context-free language;
- It can be parsed in time polynomial in the length of the input;
- It can express multiple agreement, crossed agreement and duplication;
- It has the constant growth property.

Informally, the constant growth property states that if we order the sentences of a language by increasing length, then the length of two consecutive strings do not differ by more than a constant.

The grammar formalisms TREE ADJOINING GRAMMAR (TAG; Joshi et al., 1975; Joshi and Schabes, 1997), HEAD GRAMMAR (HG; Pollard, 1984), LINEAR INDEXED GRAMMAR (LIG; Gazdar, 1987) and COMBINATORY CATEGORIAL GRAMMAR (CCG; Steedman, 1985, 1986) were all developed independently of each other, with the aim of overcoming the problems of CFG. They are all mildly context-sensitive, and were shown equivalent by Vijay-Shanker and Weir (1994).

The non-context-free constructions above can all be generalized to more complex forms;

- k-multiple agreement, $a_1^n \ldots a_k^n$;
- k-crossed agreement, $a_1^{n_1} \ldots a_k^{n_k} b_1^{n_1} \ldots b_k^{n_k}$;
- k-duplication, $\{ w^k \mid w \in (a \cup b)^* \}$;

These (and similar) general languages can be used to give bounds on the expressivity of grammar formalisms. For example, CFG can express at most 2-multiple agreement, 1-crossed agreement and 1-duplication, whereas TAG and equivalent formalisms can express at most 4-multiple agreement, 2-crossed agreement and 2-duplication. This can be extended to formalisms that can express these properties for any given k. Two such formalism are LINEAR CONTEXT-FREE REWRITING SYSTEMS (LCFRS; Vijay-Shanker et al., 1987) and LINEAR MULTIPLE CONTEXT-FREE GRAMMAR (LMCFG; Seki et al., 1991), which are still mildly context-sensitive in the sense above; where a k-LCFRS can express at most $2k$-multiple agreement, k-crossed agreement and k-duplication.

Limitations of mildly context-sensitive formalisms

However, there are limitations of a mildly context-sensitive formalism; e.g. it cannot describe the exponentially growing language a^{2^n}, simply because that language is not constantly growing. The formalisms PARALLEL MULTIPLE CONTEXT-FREE GRAMMAR (PMCFG; Seki et al., 1991), *simple* LITERAL MOVEMENT GRAMMAR (s-LMG; Groenink, 1997a,b) and RANGE CONCATENATION GRAMMAR (RCG; Boullier, 2000b,a) can describe that exponential grammar. But on the other hand PMCFG cannot describe the language $(a \cup b)^{2^n}$.

This last language can be described by context-sensitive formalisms, such as HEAD-DRIVEN PHRASE STRUCTURE GRAMMAR (HPSG; Pollard and Sag, 1994) or LEXICAL FUNCTIONAL GRAMMAR (LFG; Bresnan and Kaplan, 1982); but then there are other languages that these formalisms cannot describe, e.g. the set of all valid propositions in first-order logic. Highest in the hierarchy are the recursively enumerable languages, that can be described by Turing-complete formalisms.

1.2.2 Complexity of parsing

The standard way of describing the theoretical efficiency of algorithms is to calculate the worst-case time complexity, parameterized over the length of the input string. For this purpose we use the *ordo* notation, where $f(n) = \mathcal{O}(g(n))$ says that the function f grows at most as fast as g. For our purposes we only need to note that $n^k = \mathcal{O}(n^{k+1})$ and $n^k = \mathcal{O}(a^n)$, but that $n^{k+1} \neq \mathcal{O}(n^k)$ and $a^n \neq \mathcal{O}(n^k)$. Or in other words, polynomial functions are better than exponential, and the lower the degree the better.

Parsing of context-free grammars

Parsing of context-free grammars can be accomplished in time cubic in the length of the input string, $\mathcal{O}(n^3)$. Several different algorithms exist; the simple CKY algorithm (Kasami, 1965; Younger, 1967) has been extended by Earley

(1970), Graham et al. (1980) and Kilbury (1985), just to mention a few. These and similar algorithms are called *tabular* or *chart parsing* algorithms (Kay, 1986; Wirén, 1992).

Other algorithms compile the grammar into a *push-down automaton*. Knuth (1965) introduced the LR parsing algorithm, which has been widely used for parsing of deterministic grammars. The extension to non-deterministic grammars, such as grammars for natural languages, was made by Lang (1974) and Tomita (1986); later work has reformulated these as tabular algorithms (Lang, 1994; Nederhof and Satta, 1996).

Most context-free parsing algorithms can be given a formulation in a parsing framework, such as *parsing as deduction* (Shieber et al., 1995) or *parsing schemata* (Sikkel, 1997b).

Parsing of more expressive formalisms

The more expressive a formalism is, the higher is its parsing complexity. As an example, TAG parsing can be accomplished in time $\mathcal{O}(n^6)$, as first shown by Vijay-Shanker and Joshi (1985); and PMCFG parsing can be accomplished in time $\mathcal{O}(n^e)$, where e is a constant depending on the grammar (Seki et al., 1991). s-LMG and RCG both characterize exactly the class of languages recognizable in polynomial time (Groenink, 1997a,b; Boullier, 2000a,b). More expressive formalisms might take exponential time $\mathcal{O}(e^n)$ or may even be undecidable.

A context-free ID/LP grammar can be transformed to an equivalent CFG, thus making it parsable in cubic time. But the grammar size can explode exponentially, making the dependence on the grammar dominating. Shieber (1984) has given a direct parsing algorithm for ID/LP grammars, which reduces the overhead of parsing.

In the last years there has been interest in linearization-based HPSG grammars. When parsing these grammars, one uses bit vectors of length n to represent the input string of length n (Reape, 1991; Daniels and Meurers, 2002). This gives rise to 2^n possibilities for the bit vectors, and thus the algorithms are exponential, but on the other hand HPSG is in itself an exponential formalism.

Reducing to boolean matrix multiplication

Valiant (1975) has shown that it is possible to transform the CKY algorithm into the problem of boolean matrix multiplication (BMM), for which there are sub-cubic algorithms. The best known complexity for BMM is approximately $\mathcal{O}(n^{2.376})$, by Coppersmith and Winograd (1990).

For more expressive formalisms it is also sometimes possible to reduce the parsing problem to BMM; e.g. TAG parsing has been reduced by Rajasekaran and Yooseph

(1995) to multiplying two $n^2 \times n^2$ boolean matrices, which gives a lower complexity bound of $\mathcal{O}\left((n^2)^{2.376}\right) = \mathcal{O}(n^{4.752})$. Furthermore, Nakanishi et al. (1997, 1998) has extended the technique to give lower complexity bounds for parsing of LCFRS, LMCFG and PMCFG.

However, these sub-cubic algorithms all involve large constants making them inefficient in practice. And, since a BMM of size n can be reduced to context-free parsing of length n (Lee, 2002), and similarly a size n^2 BMM can be reduced to length n TAG parsing (Satta, 1994); there is not much hope in finding practical parsing algorithms with better time complexity than $\mathcal{O}(n^3)$ (or $\mathcal{O}(n^6)$ for TAG-equivalent formalisms).

Practical parsing algorithms

Unfortunately, apart from the TAG formalism (and the equivalent ones) and the unification-based formalisms, almost all parsing algorithms are extensions of the CKY algorithm. This algorithm has a good theoretical worst-case complexity, but performs badly in practice. The Earley algorithm and its relatives are often better choices, which is shown by the fact that most formalisms that have been used for practical purposes also have implementations of Earley-like parsers and similar algorithms.

1.2.3 Storing parse results

A string recognized by a context-free grammar might have an exponential number (in the length of the string) of syntactical analyses, which are called *parse trees*. A classical example is a grammar for mathematical expressions containing the rule,

$$\mathsf{Exp} \quad \rightarrow \quad \mathsf{Exp} \text{ ` + ' } \mathsf{Exp}$$

In some pathological cases (i.e. when the grammar is cyclic), there might even be an infinite number of trees. The polynomial parse time complexity comes from the fact that all these parse trees can be compactly stored in polynomial space, in a *parse forest*, also known as a *chart*.

Parsing as intersection

A parse forest can be represented as a context-free grammar, recognizing the language consisting of only the input string. This is a consequence of the fact that the class of context-free languages is closed under intersection with regular languages. Bar-Hillel et al. (1964) gave an algorithm for calculating the intersection, thus also giving one of the first parsing algorithms for context-free grammars. The resulting CFG directly represents all possible parse trees for the given input. The forest can then be further investigated to remove useless nodes, increase sharing and reduce space complexity (Billot and Lang, 1989), and in fact all chart parsing algorithms can be seen as variants of this idea.

More expressive formalisms

The idea of parsing as intersection has been extended to more expressive formalisms, such as TAG and LIG (Vijay-Shanker and Weir, 1990, 1993b,a) and even very general formalisms such as LCFRS (Lang, 1994). An interesting consequence of the idea is that for even more expressive formalisms "parsing" (i.e. constructing the intersection) can be easier than "recognition" (i.e. deciding whether the input is recognized); while the intersection with a regular set can be performed efficiently resulting in a parse forest, checking the forest for whether the input was recognized or extracting parse trees can be quite costly.

Still the idea can be helpful when implementing parsers for very expressive formalisms; when parsing HPSG one often removes the DAUGHTERS feature from the elements in the chart, to reduce space complexity. This feature corresponds to the parse tree of a CFG, and can always be deduced from the final chart when necessary.

1.3 Separating abstract and concrete syntax

The grammar formalisms studied in this thesis all have one thing in common; the separation of abstract and concrete syntax. The abstract part of a grammar defines a set of abstract syntactic structures, called abstract terms or trees; and the concrete part defines a relation between abstract structures and concrete structures.

The distinction between abstract and concrete syntax has been made by several authors since the late 1950's; McCarthy (1963) and Landin (1966) made the distinction in describing the syntax for programming languages; Chomsky (1957, 1965) made the distinction between (abstract) *deep structure* and (concrete) *surface structure*, together with transformations between the structures; Curry (1963) introduced the distinction under the headings of *tectogrammatic* and *phenogrammatic* structure; and Montague (1974) viewed a grammar as a set of rules linearizing logically interpreted (abstract) analysis trees into (concrete) strings of a natural language.

A linearization perspective

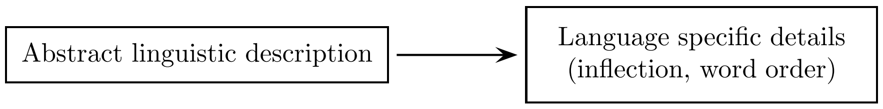
The formalisms studied in this thesis all have a linearization perspective, where the relation between abstract and concrete is viewed as a mapping from abstract to concrete structures, called *linearization terms.* In some cases the mapping can be partial or even many-valued.

1.3.1 Linguistic advantages

Although not exploited in many well-known grammar formalisms, a clear separation between abstract and concrete syntax gives some advantages.

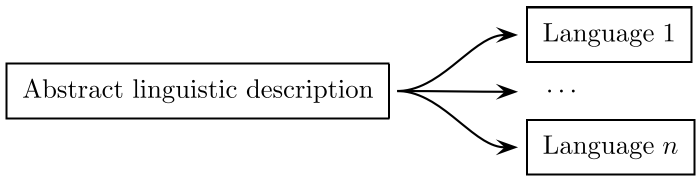
Higher-level language descriptions

The grammar writer has a greater freedom in describing the syntax for a language. When describing the abstract syntax he/she can choose not to take certain language specific details into account, such as inflection and word order. Abstracting away smaller details can make the grammars simpler, both to read and understand, and to create and maintain.


Abstract linguistic description
Language specific details (inflection, word order)

Multilingual and multimodal grammars

It is possible to define several different concrete syntax mappings for one particular abstract syntax. The abstract syntax could e.g. give a high-level description of a family of similar languages, and each concrete mapping gives a specific language instance.



This kind of multilingual grammar can be used as a model for interlingua translation between languages. But we do not have to restrict ourselves to only multilingual grammars; different concrete syntaxes can be given for different modalities. As an example, consider a grammar for displaying time table information. We can have one concrete syntax for writing the information as plain text, but we could also present the information in the form of a table output as a LaTeX file or in EXCEL format, and a third possibility is to output the information in a format suitable for speech synthesis.

Syntax editing

It is possible to write documents by directly editing the abstract syntax, and let the program display the resulting concrete syntax. This was done for programming languages in e.g. the systems MENTOR (Donzeau-Gouge et al., 1975) and CORNELL PROGRAM SYNTHESIZER (Teitelbaum and Reps, 1981); and has been generalized to natural language grammars and even *multilingual document authoring* (Dymetman et al., 2000; Khegai et al., 2003), where a document is written simultaneously in several languages. One example of multilingual authoring is when writing technical user manuals which should have exactly the same interpretation in any language.

Several descriptional levels

In this thesis we only talk about formalisms with two descriptional levels; abstract and concrete. But this can be generalized to as many levels as is wanted, by equating the concrete syntax of one grammar level with the abstract syntax of another level. As an example we could have a spoken dialogue system with a semantical, a syntactical, a morphological and a phonological level. This system has to define three mappings; *i*) a mapping from semantical descriptions to syntax trees; *ii*) a mapping from syntax trees to sequences of lexical tokens; and *iii*) a mapping from lexical tokens to lists of phonemes.

$$\begin{tikzcd} \boxed{\text{Semantics}} \arrow[r] & \boxed{\text{Syntax}} \arrow[r] & \boxed{\text{Morphology}} \arrow[r] & \boxed{\text{Phonology}} \end{tikzcd}$$

This formulation makes grammars similar to transducers (Karttunen et al., 1996; Mohri, 1997) which are mostly used in morphological analysis, but has been generalized to dialogue systems by Lager and Kronlid (2004).

Grammar composition

A multi-level grammar as described above, can be viewed as a “black box”, where the intermediate levels are unknown to the user. Then we are back in our first view as a grammar specifying an abstract and a concrete level together with a mapping. In this way we can talk about *grammar composition*, where the composition $G_2 \circ G_1$ of two grammars is possible if the abstract syntax of G_2 is equal to the concrete syntax of G_1. The result of the composition is the grammar inheriting the abstract syntax from G_1, the concrete syntax from G_2, and having the linearization mapping $f_2 \circ f_1$, where f_1, f_2 are the linearization mappings for G_1, G_2 respectively.

If the grammar formalism supports this, a composition of several grammars can be pre-compiled into a compact and efficient grammar which doesn't have to mention the intermediate domains and structures. This is the case for e.g. finite state transducers, but also for GF as has been shown by Ranta (2004b).

Resource grammars

The possibility of separate compilation of grammar compositions, opens up for writing *resource grammars* (Ranta, 2004b). A resource grammar is a fairly complete linguistic description of a specific language. Many applications do not need the full power of a language, but instead want to use a more well-behaved subset, which is often called a *controlled language*. Now, if we already have a resource grammar, we do not even have to write a concrete syntax for the desired controlled language, but instead we can specify the language by mapping structures in the controlled language into structures in the resource grammar.

$$\begin{tikzcd}\boxed{\text{Controlled syntax}} \arrow[r] & \boxed{\text{Resource syntax}} \arrow[r] & \boxed{\text{Object language}}\end{tikzcd}$$

1.3.2 Comparison with some grammar formalisms

Here we compare some existing grammar formalisms from the perspective of the ability to separate abstract and concrete syntax. We have no intention of giving a full description of the formalisms, and the reader can safely skip any part of this section. The main formalisms studied in this thesis, grammatical framework and generalized context-free grammar, are presented in the next two sections.

Context-free grammar (CFG)

A context-free grammar has no separation of abstract and concrete syntax whatsoever. There is only one level of syntax rules, defining both the abstract syntax trees and the concrete language. The concrete syntax is not structured at all, making it impossible, or at least very complicated, to have several descriptional levels.

Head grammar (HG)

Head grammar (Pollard, 1984) in an extension of cfg, where the concrete syntax is *headed strings*, which can be concatenated or *wrapped* inside another headed string. There is not much structure in the concrete syntax, and the abstract syntax is tightly connected to the concrete word order.

Categorial grammar (CG)
Combinatory categorial grammar (CCG)

Categorial grammar (Ajdukiewicz, 1935; Bar-Hillel, 1953; Lambek, 1958) is equivalent to cfg, but instead of grammar rules it has complex *functional categories*, together with rules for *function application*. Combinatory categorial grammar (Steedman, 1985, 1986) also adds rules for *function composition* to the framework, thus yielding an extension of cfg.

The notion corresponding to abstract syntax is the derivation trees, and they are tightly bound to the order of the given words. There are extensions (e.g. type logical grammar; Morrill, 1994) that add some word order freedom, but the concrete syntax is nevertheless simple strings. This means that cg and relatives are similar to cfg when it comes to separating abstract and concrete syntax.

Indexed grammar (IG)
Linear indexed grammar (LIG)

Indexed grammar (Aho, 1968) and linear indexed grammar(Gazdar, 1987) are also extensions of cfg. In these formalisms the context-free categories are augmented with a *stack of indices.* On each application of a rule, an index can be pushed onto or popped from a stack. But the abstract syntax as represented by the syntax tree is still tightly connected to the concrete syntax of strings.

Tree adjoining grammar (TAG)

Tree adjoining grammar (Joshi et al., 1975; Joshi and Schabes, 1997) is a formalism based on trees and a tree rewriting operation called *adjunction.* It shares the basic problem with cfg, that there is only one descriptional level; syntax trees are directly correlated to the concrete word order.

Linear context-free rewriting systems (LCFRS)
Parallel multiple context-free grammar (PMCFG)

Linear context-free rewriting systems (Vijay-Shanker et al., 1987) and parallel multiple context-free grammar (Seki et al., 1991) are defined as instances of gcfg where the linguistic objects are tuples of strings. The operations associated with syntax rules are only allowed to use tuple projection and string concatenation, and lcfrs has some extra restrictions on the linearization functions to ensure mild context-sensitivity. Since they are defined as gcfg, they share the same separation of abstract and concrete syntax. The only drawback is that the concrete syntax is restricted to string tuples.

Literal movement grammar (LMG)
Range concatenation grammar (RCG)

These formalisms are very similar; a grammar is seen as a collection of Horn-like clauses over predicates, just as in the programming language prolog. Groenink (1997a,b) introduced literal movement grammar, where predicates range over tuples of strings, making the formalism Turing-complete. There are also restricted variants called *simple* LMG (*s*-LMG) and RANGE CONCATENATION GRAMMAR (Boullier, 2000a,b), which characterize the class of languages recognizable in polynomial time. LMG and RCG are similar to GCFG, and share the same representation of abstract syntax. The drawbacks are that the concrete syntax is restricted to strings, and that the abstract and concrete syntax are defined simultaneously, making it difficult to use the same abstract syntax with several concrete.

Lexical functional grammar (LFG)

Lexical functional grammar (Bresnan and Kaplan, 1982) has a clean division between *c-structures* and *f-structures*; the former represents concrete syntax as trees, and the latter represents the "functional" (or abstract) structure as feature structures. Since the structures are clearly specified, it is difficult to implement several levels of abstraction; apart from that, LFG inherits all advantages of a clear separation between abstract and concrete syntax.

Dependency grammar (DG)

Dependency grammar consists of a large and diverse family of grammar formalisms, all sharing the assumption that syntactic structure consists of *lexical nodes* linked by binary relations called *dependencies* (see e.g. Mel'cuk, 1988; Hudson, 1990); meaning that DG do not have the idea of phrases. Because of the diversity it is difficult to make general comments regarding the separation of abstract and concrete syntax. There are formalisms (Hays, 1964; Gaifman, 1965) having no separation at all; and there are more recent formalisms (Debusmann et al., 2004) where the concrete syntax is not even limited to strings.

Head-driven phrase structure grammar (HPSG)

The syntactical structures in HEAD-DRIVEN PHRASE STRUCTURE GRAMMAR (Pollard and Sag, 1994) are *typed feature structures*, similar to but more powerful than records.

An HPSG grammar has several descriptional levels, for phonology, syntax, semantics etc., but the separation is not always that clear. The different levels all live together in one single feature structure, as different features. E.g. concrete strings resides under the feature PHON, whereas the syntactic structure is split into several parts. This makes it difficult to generalize HPSG to multilingual grammar, but also to perform compilation to remove intermediate levels.

Later work on linearization-based HPSG has separated the concrete word order from the feature structures (Reape, 1991; Daniels and Meurers, 2002), thus giving a better separation of concrete and abstract syntax.

1.3.3 Generalized context-free grammar

Generalized context-free grammar (gcfg) was introduced by Pollard (1984) as a mathematical framework for describing head grammar, an extension of context-free grammars. Although the main idea of Pollard was not the separation of abstract and concrete syntax, gcfg can be seen as a very nice example of this idea. Since gcfg is a very expressive grammar formalism involving general (Turing-complete) partial functions, its main usage is as a framework for specifying more restricted grammar formalisms.

A definition of a gcfg consists of a context-free grammar, where each n-ary rule

$$A \quad \to \quad f[A_1, \ldots, A_n]$$

is associated with an n-ary operation over *(linguistic) objects*,

$$f^\circ \quad \in \quad O^n \to O$$

The set O of linguistic objects is not further specified, and the n-ary operation f° can be any partial mapping from O^n to O. The context-free grammar corresponds to the abstract syntax, and the operations together with the set of linguist objects correspond to the concrete syntax.

Often it is more fruitful to view gcfg as a framework for describing formalisms, rather than a specific formalism itself. The reason is that the definitions of what constitutes an object or an operation are very vague. A grammar formalism is an instance of gcfg if the structure of O is specified, and if it describes how operations can be formed.

Instances of GCFG

The following is a list of some grammar formalisms that can be seen as relatively direct instances of gcfg.

Context-free grammar The linguistic objects are strings, and the only allowed operation is concatenation.

Head grammar The linguistic objects are strings with a distinguished *head* element, and apart from concatenation, there is also a *wrapping* operation;

Indexed grammar The linguistic objects are pairs of strings and *stacks*, and together with string concatenation, there are the usual stack operations;

Linear context-free rewriting systems The linguistic objects are *string tuples*, and the allowed operations are concatenation on the elements of the tuples;

Parallel multiple context-free grammar Similar to LCFRS, but argument strings can be deleted and duplicated at will.

Other formalisms such as TAG, CCG, LMG and RCG have slightly less intuitive formulations as instances of GCFG.

1.3.4 Grammatical Framework

The abstract theory of GRAMMATICAL FRAMEWORK (GF; Ranta, 2004a) is a version of dependent type theory, similar to LF (Harper et al., 1993), ALF (Magnusson and Nordström, 1994) and COQ (Coq, 1999). What GF adds to the logical framework is a possibility to define concrete syntax, that is, notations expressing formal concepts in user-readable ways. In this sense GF fits well into the idea of separating abstract and concrete syntax.

The development of GF started as a notation for TYPE-THEORETICAL GRAMMAR (Ranta, 1994), which use Martin-Löf's type theory (1984) to express the semantics of natural language. The development of GF as an authoring system started as a plug-in to the proof editor ALF, to permit natural-language rendering of formal proofs (Hallgren and Ranta, 2000). The extension of the scope outside mathematics was made in the Multilingual Document Authoring project at XEROX (Dymetman et al., 2000). In continued work, GF has been used in areas like software specifications (Hähnle et al., 2002) and dialogue systems (Ranta and Cooper, 2004).

After the first publication (Mäenpää and Ranta, 1999), the expressiveness of the concrete syntax has developed into a functional programming language. As such it is similar to a restricted version of programming languages like HASKELL (Peyton Jones, 2003) and ML (Milner et al., 1997). The language is restricted enough to be possible to compile into an efficient canonical format, but expressive enough to incorporate modern programming language constructs such as user-definable data types, higher-order functions, and a module system for defining grammatical resources.

Type theory

The abstract syntax of a GF grammar is defined by declaring a number of basic types (called *categories*), and a number of basic functions. A function is declared by giving its *typing*,[2]

$$f \quad : \quad B_1 \times \cdots \times B_\delta \to A$$

[2]Note that the notation for GF we use is different from the notation used in the actual GF implementation and in other publications; the differences are spelled out in section 2.3.8.

This declaration states that f is a function taking δ arguments of types $B_1, \ldots, B_\delta$, resulting in a term of type A. A function with no arguments ($\delta = 0$) is called a *constant*, and is simply declared as,

$$c \quad : \quad A$$

In general we write $t : T$ if the term t is of type T. By applying the basic functions to each other, compound terms can be formed,

$$f(c_1, \ldots, c_\delta) \quad : \quad A$$

whenever each $c_i : B_i$ and f is declared as above.

Higher-order functions and dependent types

It is also possible to declare higher-order functions and dependent types in a GF grammar. A higher-order function is a function where some of the arguments are functions themselves; and a dependent type is declared to depend on (one or more) terms of other types.

These features are more thoroughly described in section 2.3.1, but they are not used until in chapter 6 of this thesis. Instead we concentrate on the very important subclass *context-free* GF, which does not contain higher-order functions or dependent types.

Concrete linearizations

The novel thing about GF with respect to a logical framework, is that it adds a mapping from abstract terms to concrete *linearizations*. To define a concrete syntax of a grammar, we only need to do the following.

- For each basic category A defined in the abstract syntax, we define a corresponding *linearization type* A°.
- For each basic function f defined in the abstract syntax, we define a corresponding *linearization function* f°. If the original function f has a typing,

  $$f \quad : \quad B_1 \times \cdots \times B_\delta \rightarrow A$$

  then the linearization function f° has the typing,

  $$f^\circ \quad : \quad B_1^\circ \times \cdots \times B_\delta^\circ \rightarrow A^\circ$$

The linearization of a term $t : T$ can now be defined as $[\![t]\!] = f^\circ([\![t_1]\!], \ldots, [\![t_\delta]\!])$ whenever $t = f(t_1, \ldots, t_\delta)$. The constraints on the linearization definitions assure that linearizations always have the correct type. Grammars are thus *compositional* in the sense that a linearization is a function of the argument linearizations, not of the arguments themselves.

The module system

GF has a module system, inspired by ideas from programming languages. There are three kinds of modules; abstract, concrete and resource modules.

- An abstract module defines an abstract theory, with categories and functions.
- A concrete module defines the concrete syntax of an abstract theory, by giving linearization types and linearization functions.
- A resource module defines parameter types, and operations that can be used as helper functions in concrete modules.

Modules can *extend* other modules by adding new definitions, thus opening the possibilities for modular grammar engineering. Another useful feature is that a concrete module (together with the corresponding abstract module) can be translated into a resource module. Since a resource module can be used by another concrete module, this makes it possible to perform grammar compositions as described in section 1.3.1.

1.3.5 An introductory example: Transforming a context-free grammar into GF

In this section we give some examples of how to write grammars in GF, just to get a feeling of the possibilities.

We start with a simple context-free grammar for a fragment of English. It consists of the context-free categories S, NP, VP, D, N and V (standing for Sentence, Noun Phrase, Verb Phrase, Determiner, Noun and Verb respectively), and has the following rules;

$$\begin{array}{rcllcl}
\mathsf{S} & \rightarrow & \mathsf{NP}\ \ \mathsf{VP} & & & \\
\mathsf{NP} & \rightarrow & \mathsf{D}\ \ \mathsf{N} & & & \\
\mathsf{NP} & \rightarrow & \mathsf{N} & & & \\
\mathsf{VP} & \rightarrow & \mathsf{V}\ \ \mathsf{NP} & & & \\
\mathsf{D} & \rightarrow & \text{`}\textit{a}\text{'} & & & \\
\mathsf{D} & \rightarrow & \text{`}\textit{many}\text{'} & & & \\
\mathsf{N} & \rightarrow & \text{`}\textit{lion}\text{'} & \mid & \text{`}\textit{lions}\text{'} \\
\mathsf{N} & \rightarrow & \text{`}\textit{fish}\text{'} & & & \\
\mathsf{V} & \rightarrow & \text{`}\textit{eats}\text{'} & \mid & \text{`}\textit{eats}\text{'}
\end{array}$$

The abstract syntax

To get a corresponding GF grammar, we start by giving the abstract syntax. First we have to give a name to each of the CFG rules, and then we can introduce the type declarations,

$$\begin{aligned}
s_p &: \mathsf{NP} \times \mathsf{VP} \to \mathsf{S} \\
np_d &: \mathsf{D} \times \mathsf{N} \to \mathsf{NP} \\
np_p &: \mathsf{N} \to \mathsf{NP} \\
vp_t &: \mathsf{V} \times \mathsf{NP} \to \mathsf{VP} \\
d_a,\, d_m &: \mathsf{D} \\
n_c,\, n_f &: \mathsf{N} \\
v_e &: \mathsf{V}
\end{aligned}$$

The predication function s_p forms a sentence out of a noun phrase and a verb phrase. There are two ways of forming noun phrases; either by a determiner and a noun (‘*a lion*’, ‘*many lions*’), or just a plural noun (‘*lions*’). We assume that all verbs are transitive, so we only have the transitive verb phrase forming function vp_t. The determiners d_a, d_m are singular and plural indefinites (‘*a*’ and ‘*many*’); n_c, n_f are the nouns ‘*lion*’ and ‘*fish*’; and v_e is the verb ‘*eat*’.

The concrete syntax

If we only want a GF grammar that is equivalent to the original CFG, we can assign the same linearization type to each category, $\mathsf{S}^\circ = \mathsf{NP}^\circ = \cdots = \{\, s : \mathsf{Str} \,\}$, which is a record consisting of only one string.[3] The concrete linearizations then look like follows,[4]

$$\begin{aligned}
s_p^\circ(x,\, y) &= \{\, s = x.s \cdot y.s \,\} \\
np_d^\circ(x,\, y) &= \{\, s = x.s \cdot y.s \,\} \\
np_p^\circ(x) &= \{\, s = x.s \,\} \\
vp_t^\circ(x,\, y) &= \{\, s = x.s \cdot y.s \,\} \\
d_a^\circ &= \{\, s = \text{‘}a\text{’} \,\} \\
d_m^\circ &= \{\, s = \text{‘}many\text{’} \,\} \\
n_c^\circ &= \{\, s = \text{‘}lion\text{’} \quad | \quad \text{‘}lions\text{’} \,\} \\
n_f^\circ &= \{\, s = \text{‘}fish\text{’} \,\} \\
v_e^\circ &= \{\, s = \text{‘}eats\text{’} \quad | \quad \text{‘}eat\text{’} \,\}
\end{aligned}$$

[3]The reason for using records and not just strings will become apparent later.

[4]The alert reader might notice that we abuse notation somewhat here, by using the non-deterministic choice (|) which is an extension introduced in section 5.2, but the grammar will anyway be improved upon later.

A concrete syntax that takes care of agreement

If we want to change the grammar so that it also takes care of agreement, we can do as follows. First we introduce the parameter type Num with the two values or constructors Sg and Pl;

$$\textbf{param}\ \mathsf{Num} \quad = \quad \mathsf{Sg} \mid \mathsf{Pl}$$

Then we make a decision that nouns, verbs and verb phrases are *parameterized* over the number; whereas determiners and noun phrases have an *inherent* number.[5] A phrase parameterized over P is stored as an inflection table $P \Rightarrow \mathsf{Str}$; and an inherited parameter is stored in a record together with the linearized string,

$$\begin{aligned} \mathsf{N}^\circ = \mathsf{V}^\circ = \mathsf{VP}^\circ &= \{\, s : \mathsf{Num} \Rightarrow \mathsf{Str} \,\} \\ \mathsf{D}^\circ = \mathsf{NP}^\circ &= \{\, s : \mathsf{Str}\,;\ n : \mathsf{Num} \,\} \end{aligned}$$

To give the value of an inherent parameter, we simply form a record; and to access the value of an inherent parameter, we use record projection (just as we do to access the linearized string). An inflection table is formed by $[\, p_1 \Rightarrow t_1\,;\ \ldots\,;\ p_n \Rightarrow t_n \,]$, where $p_1, \ldots, p_n$ are inflection *patterns*; and to apply an inflection table to a parameter, we use the *selection* operation (!). Returning to our example, we get the following concrete syntax for the English grammar with number agreement between the subject and the verb,[6]

$$\begin{aligned} s_p^\circ(x,\, y) &= \{\, s = x.s \cdot y.s \,!\, x.n \,\} \\ np_d^\circ(x,\, y) &= \{\, s = x.s \cdot y.s \,!\, x.n\,;\ n = x.n \,\} \\ np_p^\circ(x) &= \{\, s = x.s \,!\, \mathsf{Pl}\,;\ n = \mathsf{Pl} \,\} \\ vp_t^\circ(x,\, y) &= \{\, s = [\, z \Rightarrow x.s \,!\, z \cdot y.s \,] \,\} \\ d_a^\circ &= \{\, s = \text{‘}\textit{a}\text{’}\,;\ n = \mathsf{Sg} \,\} \\ d_m^\circ &= \{\, s = \text{‘}\textit{many}\text{’}\,;\ n = \mathsf{Pl} \,\} \\ n_c^\circ &= \{\, s = [\, \mathsf{Sg} \Rightarrow \text{‘}\textit{lion}\text{’}\,;\ \mathsf{Pl} \Rightarrow \text{‘}\textit{lions}\text{’} \,] \,\} \\ n_f^\circ &= \{\, s = [\, \_ \Rightarrow \text{‘}\textit{fish}\text{’} \,] \,\} \\ v_e^\circ &= \{\, s = [\, \mathsf{Sg} \Rightarrow \text{‘}\textit{eats}\text{’}\,;\ \mathsf{Pl} \Rightarrow \text{‘}\textit{eat}\text{’} \,] \,\} \end{aligned}$$

Note that the table in vp_t° has only one pattern matching any parameter, binding it to the variable z which can be used in the table body. Also note that the table in n_f° has an *anonymous* pattern, meaning that the value is ‘*fish*’ regardless of

[5] GF has a functional perspective on linearizations, meaning that parameters have to be either parameterized over or inherited. The principal way of making parameters agree is to apply a parameterized inflection table to an inherited parameter.

[6] A notational convention throughout this thesis is that record projection (.) binds harder than table selection (!), which in turn binds harder that concatenation ($\cdot$).

the inflection parameter. Both uses are examples of that tables can sometimes be compacted.

Examples of phrases that are disallowed by this concrete syntax are *i*) noun phrases consisting of just a singular noun; *ii*) noun phrases where the determiner and noun does not agree; and *iii*) sentences where the subject noun phrase does not agree with the following verb.

This English grammar will be used as the main example grammar in this thesis; the grammar is also shown in figure 2.3 on page 49.

A concrete syntax for Swedish

Swedish has a more complex morphology than English; nouns do not only depend on number, they also have an inherent *gender* (neuter and uter) associated to them. Determiners, on the other hand, have number as an inherent feature and depend on the gender of the noun. First we have to declare the corresponding parameter type Gen;

$$\textbf{param}\ \mathsf{Gen} \quad = \quad \mathsf{Neu} \mid \mathsf{Utr}$$

then the linearization types for nouns and determiners can be declared as,

$$\begin{aligned} \mathsf{N}^\circ &= \{\, s : \mathsf{Num} \Rightarrow \mathsf{Str}\,;\ g : \mathsf{Gen}\,\} \\ \mathsf{D}^\circ &= \{\, s : \mathsf{Gen} \Rightarrow \mathsf{Str}\,;\ n : \mathsf{Num}\,\} \end{aligned}$$

Now we can define the linearizations for the determiners d_a, d_m and the nouns n_c, n_f;

$$\begin{aligned} d_a^\circ &= \{\, s = [\,\mathsf{Utr} \Rightarrow \text{‘}\textit{en}\text{’}\,;\ \mathsf{Neu} \Rightarrow \text{‘}\textit{ett}\text{’}\,]\,;\ n = \mathsf{Sg}\,\} \\ d_m^\circ &= \{\, s = [\,\_ \Rightarrow \text{‘}\textit{många}\text{’}\,]\,;\ n = \mathsf{Pl}\,\} \\ n_c^\circ &= \{\, s = [\,\_ \Rightarrow \text{‘}\textit{lejon}\text{’}\,]\,;\ g = \mathsf{Neu}\,\} \\ n_f^\circ &= \{\, s = [\,\mathsf{Sg} \Rightarrow \text{‘}\textit{fisk}\text{’}\,;\ \mathsf{Pl} \Rightarrow \text{‘}\textit{fiskar}\text{’}\,]\,;\ g = \mathsf{Utr}\,\} \end{aligned}$$

Noun phrases, on the other hand, do not influence the inflection of verbs, which means that they can have simple linearization types $\mathsf{NP}^\circ = \mathsf{V}^\circ = \{\, s : \mathsf{Str}\,\}$.[7] Now we are ready to give the linearization functions for noun phrase forming;

$$\begin{aligned} np_d^\circ(x,\, y) &= \{\, s = x.s\,!\,y.g \cdot y.s\,!\,x.n\,\} \\ np_p^\circ(x) &= \{\, s = x.s\,!\,\mathsf{Pl}\,\} \end{aligned}$$

Finally, the word order of sentences depend on the context of the sentence. There are three different word orders (direct, indirect and subordinate), introducing yet another parameter type Order;

$$\textbf{param}\ \mathsf{Order} \quad = \quad \mathsf{Dir} \mid \mathsf{Indir} \mid \mathsf{Sub}$$

[7]This is a simplification; when adding pronouns and/or adjectives, Swedish noun phrases can get quite complex.

The indirect order (used e.g. in questions) puts the subject noun phrase inside the verb phrase. The way to solve this in GF is to use discontinuous verb phrases. The linearization of sentences and verb phrases will be,[8]

$$\begin{aligned}
\mathsf{S}^\circ &= \{\, s : \mathsf{Order} \Rightarrow \mathsf{Str} \,\} \\
\mathsf{VP}^\circ &= \{\, s_1 : \mathsf{Str}\,;\ s_2 : \mathsf{Str} \,\} \\
s_p^\circ(x,\, y) &= \{\, s = [\,\mathsf{Indir} \Rightarrow y.s_1 \cdot x.s \cdot y.s_2\,;\ \_ \Rightarrow x.s \cdot y.s_1 \cdot y.s_2\,]\,\} \\
vp_t^\circ(x,\, y) &= \{\, s_1 = x.s\,;\ s_2 = y.s \,\}
\end{aligned}$$

A fourth possible word order could be *topicalized*, which is used when the object is put in front of the sentence for focusing purposes; e.g. the sentence '*fiskar äter många lejon*' (fish eat many lions) have the preferred reading (it is fish that many lion eat). This can be solved by adding a new constructor Top to the type Order, and a new row to the s_p° table, $[\,\mathsf{Top} \Rightarrow y.s_2 \cdot x.s \cdot y.s_1\,]$.

1.4 Overview and main results of the thesis

Here we give an overview of the thesis, together with the main results. The overview and results are presented chapter by chapter.

Chapter 2: Background

This chapter gives the theoretical background for the rest of the thesis. Grammatical framework (GF; Ranta, 2004a) is defined together with its important subclass *context-free* GF (*cf*-GF). Generalized context-free grammar (GCFG; Pollard, 1984) is introduced as a framework for describing other grammar formalisms; one instance is parallel multiple context-free grammar (PMCFG; Seki et al., 1991), which is known to have polynomial parsing complexity.

Some direct consequences of the definitions are noted; *cf*-GF is an instance of GCFG, and PMCFG is an instance of *cf*-GF.

For parsing purposes, the representation of syntactical terms is discussed. We extend the notion of a shared forest for compactly representing a set of syntactical analyses, to the GCFG formalism. We also discuss when a grammar formalism, for which there are known parsing algorithms, can be used to parse grammars in another formalism.

Chapter 3: Reducing context-free GF to PMCFG

This chapter shows that *cf*-GF is strongly equivalent to PMCFG. This equivalence is shown by giving an algorithm converting *cf*-GF grammars into PMCFG

[8]The difference between direct and subordinate word order only shows up in the presence of negation, which we don't have in this example.

grammars recognizing the same language; and by showing that parse results can be converted back efficiently.

The conversion algorithm consists of enumerating all parameter instantiations in a linearization, and then moving the instantiated parameters to the abstract categories. Enumerating all instantiations may lead to an exponential increase of the grammar size. Therefore two alternative conversion algorithms are given, which do not enumerate all possible instantiations, but instead try to only instantiate when it is necessary.

Chapter 4: Parsing algorithms for context-free GF and PMCFG

This chapter investigates a number of tabular parsing algorithms for *cf*-GF and PMCFG, all with polynomial time complexity. Starting with a general passive algorithm similar to the one given by Seki et al. (1991), several different modifications are suggested.

The search space can be reduced by approximating the PMCFG grammar by an over-generating CFG. Afterwards the context-free parse results can be translated back into PMCFG parse results, which have to be checked for correctness since the CFG is over-generating.

Another alternative is to use an active algorithm, in the spirit of the context-free Earley (1970) algorithm. We give two active algorithms; one recognizing the linearization rows of a rule in a fixed order, and another recognizing rows incrementally according to the order in which they occur in the input. Both top-down and bottom-up prediction strategies are investigated.

All suggested algorithms, except for the last incremental version, require that the PMCFG grammar is nonerasing; therefore we give an algorithm for removing erasingness from a grammar.

Chapter 5: Extensions of concrete syntax

This chapter describes four possible extensions of GF, *cf*-GF and PMCFG. Apart from investigating the resulting expressive power and parsing complexity, we also give active parsing algorithms for each of the extensions.

The intersection operation, borrowed from CONJUNCTIVE GRAMMAR (Okhotin, 2001), make PMCFG equivalent to *simple* LITERAL MOVEMENT GRAMMAR (Groenink, 1997a,b) and RANGE CONCATENATION GRAMMAR (Boullier, 2000a,b). As a corollary we get that conjunctive PMCFG describe exactly the class of languages recognizable in polynomial time.

The disjunction operation can have two possible interpretations; one intensional which does not change the descriptive power of *cf*-GF and PMCFG, and one extensional which is conjectured to be a strict extension. With extensional disjunction it is possible to describe the language $(a \cup b)^{2^n}$, which is conjectured cannot be described by *cf*-GF and PMCFG.

The third operation is the interleaving operation, which is borrowed from PARTIALLY ORDERED MULTISET CONTEXT-FREE GRAMMAR (*poms*-CFG; Nederhof et al., 2003) which in turn is a variant of the ID/LP formalism (Shieber, 1984). This operation can be reduced to a number of disjunctions, but this reduction can lead to an exponential increase of the grammar size. We instead give a direct parsing algorithm derived from a parsing algorithm for *poms*-CFG.

Chapter 6: Non-context-free abstract syntax

This final chapter discusses how to handle GF grammars containing higher-order functions or dependent types.

We give an algorithm for converting higher-order functions into first-order functions. The resulting *cf*-GF grammar is over-generating, since it cannot type-check variable occurrences correctly. We therefore give a procedure for filtering out non-well-formed terms during the conversion from first-order to higher-order parse results.

In the presence of dependent types it is possible to describe undecidable languages (Ranta, 2004a), so the parsing problem is undecidable in general. We nevertheless describe a two-step parsing process for such grammars; first we translate into an overgenerating *cf*-GF grammar, and parse using that grammar. The resulting parse items are then converted into a logic program, which can be solved by any proof search procedure.

Chapter 2

Background

This chapter gives the theoretical background for the rest of the thesis. *Grammatical Framework* (GF; Ranta, 2004a) is defined together with its important subclass *context-free* GF. *Generalized context-free grammar* (GCFG; Pollard, 1984) is introduced as a framework for describing other grammar formalisms; one instance is *parallel multiple context-free grammar* (PMCFG; Seki et al., 1991), which is known to have polynomial parsing complexity.

Some direct consequences of the definitions are noted; context-free GF is an instance of GCFG, and PMCFG is an instance of context-free GF.

For parsing purposes, the representation of syntactical terms is discussed. We extend the notion of a shared forest for compactly representing a set of syntactical analyses, to the GCFG formalism. We also discuss when a grammar formalism, for which there are known parsing algorithms, can be used to parse grammars in another formalism.

2.1 Preliminary definitions

2.1.1 Sequences, languages and grammars

Sequences

A *sequence* $x_1 \ldots x_n$ over a set X is an element of X^*, whenever each $x_i \in X$. The empty sequence is written ϵ. Concatenation is an associative operation on sequences defined as

$$x_1 \ldots x_n \cdot y_1 \ldots y_m \quad = \quad x_1 \ldots x_n y_1 \ldots y_m$$

Mathematically, concatenation and the empty sequence together form a *monoid* over X^*. This means among other things that ϵ is a *zero* for concatenation, or $\epsilon \cdot \vec{x} = \vec{x} \cdot \epsilon = \vec{x}$. When no confusion can arise, we write the concatenation $x \cdot y$ simply as xy. The *repetition* x^n is defined as n successive concatenations of x, where $x^0 = \epsilon$;

$$x^n \quad = \quad \overbrace{x \cdot \cdots \cdot x}^{n \text{ times}}$$

Apart from writing a sequence as $x_1 \ldots x_n$, it can also be written with small Greek letters, α, β, $\ldots$; or as a *vector* $\vec{x}$. In the latter case we implicitly assume that $\vec{x} = x_1 \ldots x_n$, meaning that we can use x_i as a reference to the ith element in the vector. We also use the term *strings* for sequences over an *alphabet*, where the alphabet is a finite set usually written Σ.

As a shorthand for a sequence of applications of a given function or relation, we often write $R(\vec{x}, \vec{y}, \vec{z})$ instead of $R(x_1, y_1, z_1), \ldots, R(x_n, y_n, z_n)$. Note that this presupposes that the sequences $\vec{x}$, $\vec{y}$ and $\vec{z}$ all have the same length.

Languages

A *language* is a set of strings over an alphabet. Concatenation and repetition are lifted to languages is the standard way, $AB = \{ xy \mid x \in A, y \in B \}$ and $A^n = \{ x^n \mid x \in A \}$. The *Kleene star* A^* is the union of all possible repetitions;

$$A^* \quad = \quad \bigcup_0^\infty A^i$$

When specifying a language we can identify a string s with the singleton language $\{ s \}$. All integer repetition variables are assumed to be universally quantified over. This allows us to specify languages through a regular-expression-like syntax; e.g.

$$\begin{aligned} a^n b^* a^n b^* &= \{ a^n b^i a^n b^j \mid n, i, j \geq 0 \} \\ (a \cup b)^{2^n} &= \{ w \in \{ a, b \}^* \mid |w| = 2^n, n \geq 0 \} \end{aligned}$$

Context-free grammars

We finally give the standard definition of context-free grammars.

Definition 2.1 (CFG). A *context-free grammar* is a 4-tuple $(\mathcal{C}, S, \Sigma, \mathcal{R})$, where $\mathcal{C}$ and Σ are finite sets of categories and terminals respectively, $S \in \mathcal{C}$ is the starting category, and $\mathcal{R} \subseteq \mathcal{C} \times (\mathcal{C} \cup \Sigma)^*$ is a finite set of context-free syntax rules.

Instead of writing $(B, \beta) \in \mathcal{R}$, we use the more readable $B \to \beta$. The rewriting relation $\Rightarrow$ is defined on sequences of categories and terminals, as $\alpha B \gamma \Rightarrow \alpha\beta\gamma$ whenever $B \to \beta$. The reflexive and transitive closure $\Rightarrow^*$ is used to specify the language associated with a category,

$$\mathcal{L}(A) \quad = \quad \{\, w \in \Sigma^* \mid A \Rightarrow^* w \,\}$$

The language recognized by a grammar G is $\mathcal{L}(G) = \mathcal{L}(S)$, where S is the starting category of G.

2.1.2 Data types and elements

In this thesis we talk a lot about types. We use that term in two different ways, separable by the context:

- As inductively defined types, as is used in dependent type theory. We will only use this in the abstract syntax of GF grammars, where we often call the types *categories*.

  Defining a type in this way consists of giving inference rules saying when something is a type and when something is a term of a given type. The statement $t : T$ is true when we can deduce from the inference rules that T is a type and t is of type T.

- The second usage is simply as a set of terms. We do not present a particular set theory, since we will only use simple sets; the most complex sets we use are enumerable. Defining a type in this way consists of giving the corresponding set; the statement $t : T$ is true when $t \in T$.

In the rest of this section we talk about types in the second sense.

Operations and computations

We can define operations on different types. An operation is defined by saying what types the arguments and the result should be, and by giving a computation rule of the operation. An operation can sometimes be partial, or even non-deterministic, also called many-valued.

Basic types

The type Str of *strings* is the set Σ^* of sequences over a finite set of tokens, where the token set Σ is defined in the context. Concatenation is an operation on strings defined in the standard way. A concatenation of two strings is written $s_1 \cdot s_2$, or often simply $s_1\, s_2$.

The type $\mathbb{N}$ of *natural numbers* consists of all integers ≥ 0. Addition is an operation on natural numbers.

Finally, any finite set can be seen as a type. An example is the n-element type $\mathbb{N}_n = \{\, 0, \ldots, n-1 \,\}$ of all natural numbers less than n.

Records

A *label* is an atomic symbol, not being a term or a type. If $r_1, \ldots, r_n$ are distinct labels and $T_1, \ldots, T_n$ are types, then

$$\{\, r_1 : T_1 \,;\, \ldots \,;\, r_n \,;\, T_n \,\}$$

is a record type consisting of all records,

$$\{\, r_1 = \phi_1 \,;\, \ldots \,;\, r_n = \phi_n \,\}$$

such that $\phi_i : T_i$ for $1 \leq i \leq n$. Note that the order between the rows in a record is not significant; meaning that a record is equivalent to a set of label-value pairs. Record projection is an operation taking a record and a label, defined as

$$\{\, \ldots \,;\, r = \phi \,;\, \ldots \,\}.r \quad = \quad \phi$$

Two record types $\{\, r_1 : T_1 \,;\, \ldots \,;\, r_n : T_n \,\}$ and $\{\, r'_1 : T'_1 \,;\, \ldots \,;\, r'_n : T'_n \,\}$ are equivalent, if T_i is equivalent to T'_i for each $1 \leq i \leq n$, modulo permutations of the rows.

A tuple can be seen as syntactic sugar for a record,

$$\langle \phi_1, \ldots, \phi_n \rangle \quad \equiv \quad \{\, \mathsf{1} = \phi_1 \,;\, \ldots \,;\, \mathsf{n} = \phi_n \,\}$$

where the tuple projection $\pi_i(\phi)$ is syntactic sugar for $\phi.\mathsf{i}$. Then a record $\phi = \{r_1 = \phi_1 \,;\, \ldots \,;\, r_n = \phi_n\}$ is equivalent to a tuple $\psi = \langle \phi_1 \,;\, \ldots \,;\, \phi_n \rangle$, by replacing each record projection $\phi.r_i$ by the corresponding tuple projection $\pi_i(\psi)$.

Tables

A *pattern* for a finite type P is a set of terms, or equivalently a subset of P. The patterns $p_1, \ldots, p_n$ are exhaustive for P if $p_1 \cup \ldots \cup p_n = P$. A pattern p matches a term t if $t \in p$. When writing patterns of type P we often write _ for the set of all possibilities, i.e. the full set P.

If P is a finite type and T is a type, then $P \Rightarrow T$ is a table type. The elements are tables of the form

$$[\, p_1 \Rightarrow \phi_1 \,;\, \dots \,;\, p_n \Rightarrow \phi_n \,]$$

where each p_i, $1 \leq i \leq n$, is a pattern for type P, and the patterns $p_1, \dots, p_n$ are exhaustive for P. Selection is an operation taking a table of type $P \Rightarrow T$ and a term of type P, returning a term of type T, defined as

$$[\, \dots \,;\, p \Rightarrow \phi ;\, \dots \,] \,!\, t \quad = \quad \phi$$

if p is the first pattern in the table that matches t.

A pattern is *constant* or *instantiated* if it is a singleton set. If all patterns are instantiated, then there are as many patterns as the size of P, and the table is called instantiated.

Comparing records and instantiated tables

An instantiated table is very similar to a record. The difference does not lie in the formation of tables and records, but in the associated projection and selection operations. The main difference is that for records the labels are not terms of some type. Tables on the other hand, take arguments which are of a (finite) type. So, for a record projection it is always known at compile-time which row in the record is meant, but a table can be selected by a variable and thus is not known until the variable is bound to some value.

The similarity has the effect that if all table selections in a term are instantiated, the term can be converted to another where tables are converted to records and table selections are converted to record projections.

Record unification

Records can be *unified*, which is a partial operation defined as $\Gamma_1 \sqcup \Gamma_2 = \Gamma_1 \cup \Gamma_2$ whenever there is no r such that $\Gamma_1.r \neq \Gamma_2.r$. Note that this definition is very simplistic, and not as general as the standard definitions of unification (Robinson, 1965); as an example, the definition is not recursive and thus does not unify records recursively.

Records, subrecords and flattened records

A record $\{\, r_1 = \phi_1 \,;\, \dots \,;\, r_n = \phi_n \,\}$ is equivalent to a finite set of n pairs of labels and terms. Therefore we sometimes view a record as a set, to be able to form subrecords. E.g. given a record R and a predicate P on record labels, we can form the subrecord of all rows matching P,

$$\{\, r = \phi \in R \mid P(r) \,\} \quad \subseteq \quad R$$

Testing whether a record R is a subrecord of another record R' amounts to testing $R \subseteq R'$. Note that being a subrecord is the opposite of being a subtype; the empty record is a subrecord of all records, but any record type is a subtype of the empty record type.

A nested record can be *flattened* by repeated application of the following equivalence,

$$\begin{aligned} &\{\, \ldots\,;\ r_i = \{\, r_{i1} = \phi_{i1}\,;\ \ldots\,;\ r_{in} = \phi_{in}\,\}\,;\ \ldots\,\} \\ \equiv\ \ &\{\, \ldots\,;\ r_i.r_{i1} = \phi_{i1}\,;\ \ldots\,;\ r_i.r_{in} = \phi_{in}\,;\ \ldots\,\} \end{aligned}$$

2.2 Parsing as deduction

Most parsing algorithms can be seen as a deductive process, with axioms, goals and inference rules. In this thesis we use the framework called *deductive parsing* by Shieber et al. (1995). Another wide-spread framework is *parsing schemata* by Sikkel (1997b), which could be used instead.

According to Shieber et al. (1995), parsing is "a deductive process in which rules of inference are used to derive statements about the grammatical status of strings from other such statements". The statements are called *items*, and are represented by formulae in some formal language. The inference rules and axioms are written in natural deduction style, and they can have side conditions mentioning e.g. grammar rules. The inference rules and axioms are rule schemata, meaning that they contain metavariables to be instantiated by appropriate terms when the rule is invoked. The set of items built in the deductive process is sometimes called a *chart*.

The general form of an inference rule is

$$\frac{\theta_1 \quad \ldots \quad \theta_n}{\theta} \left\{ \begin{array}{c} c_1 \\ \ldots \\ c_m \end{array} \right.$$

where θ, θ_1, …, θ_n are items and c_1, …, c_m are side conditions.

2.2.1 Soundness and completeness of algorithms

We write items as syntactic terms (e.g. $[\,R\,;\,\Gamma \bullet \phi\,;\,\vec{\Gamma}\,]$) and give an interpretation to each term. The interpretation states whether an item is grammatical given a certain input string.

Following Sikkel (1998), we prove correctness by first guessing a set of *valid items*, and then proving soundness and completeness for all items in that set.

Soundness A parsing system is *sound* if all derived items are grammatical according to the interpretation. To show soundness we only have to prove that each inference rule yields valid items whenever the antecedents are valid items.

Completeness A parsing system is *complete* if all grammatical items are derived, i.e. that we do not miss any interpretations. Completeness is often more difficult to show than soundness; but often it amounts to associating each valid item θ with a natural number $d(\theta)$ such that there is some instance of an inference rule,

$$\frac{\theta_1 \quad \ldots \quad \theta_k}{\theta} \; \{ \; \mathcal{C}$$

such that $\theta_1, \ldots, \theta_k$ are valid items, the side condition $\mathcal{C}$ holds, and $d(\theta_i) < d(\theta)$ for $1 \leq i \leq k$.

Sikkel (1998) calls the function d a *deduction length function*, while Shieber et al. (1995) use the term *rank* for $d(\theta)$. Completeness of the inference rules follows from induction on the ranks of the valid items.

2.2.2 Examples of context-free parsing algorithms

Here we give examples of some well-known parsing algorithms for context-free grammars. First we give a very simplistic algorithm, and then two refinements; the top-down algorithm of Earley (1970), and the bottom-up algorithm of Kilbury (1985). The algorithms are slightly modified for presentation purposes, but their essence are still the same. The first basic algorithm is also proved to be sound and complete. When developing active parsing algorithms for GF and PMCFG in sections 4.4 and 4.6 we do this by extending the algorithms given here.

Parse items

In these algorithms we assume that the input string is,

$$w \quad = \quad w_1 \ldots w_n$$

A substring $w_{i+1} \ldots w_j$ is said to *span the positions* $i - j$, so the whole input string w spans the positions $0 - n$.

The parse items are of the form $[\, i - j \,;\, A \rightarrow \alpha \bullet \beta \,]$ where $A \rightarrow \alpha\beta$ is a context-free rule, and $0 \leq i \leq j \leq n$ are positions in the input string. The meaning is that α is recognized spanning $i - j$; i.e. $\alpha \Rightarrow^* w_{i+1} \ldots w_j$. If β is empty the item is called *passive*. We write $[\, i - j \,;\, A \,]$ for any passive item $[\, i - j \,;\, A \rightarrow \alpha \bullet \,]$.

The goal of the parsing process is to deduce an item representing that the starting category is found spanning the whole input string; such an item can be written $[\, 0 - n \,;\, S \,]$ in our notation.

A basic context-free chart parsing algorithm

Our first context-free chart parsing algorithm consists of three inference rules. The first two, Combine and Scan, remain the same in all variants of chart parsing (sometimes only slightly modified); while the third, Predict, is a very simplistic variant, which will be improved upon later. The algorithm is also presented by Sikkel (1997b,a, 1998), who calls it "bottom-up Earley".

Combine

$$\frac{[\,i-j\,;\;A\rightarrow\alpha\bullet B\,\beta\,]\quad[\,j-k\,;\;B\,]}{[\,i-k\,;\;A\rightarrow\alpha\,B\bullet\beta\,]}\tag{2.1}$$

The basis for all chart parsing algorithms is *the fundamental rule*; saying that if there is an active item looking for a category B spanning $i-j$, and there is a passive item for B spanning $j-k$, then the dot in the active item can be moved forward, and the new item will span the positions $i-k$.

Scan

$$\frac{[\,i-j\,;\;A\rightarrow\alpha\bullet w_k\,\beta\,]}{[\,i-k\,;\;A\rightarrow\alpha\,w_k\bullet\beta\,]}\;\left\{\;k=j+1\right.\tag{2.2}$$

If the active item is looking for a terminal, then we can move the dot forward whenever the terminal is the next input token.

Predict

$$\frac{}{[\,i-i\,;\;A\rightarrow\;\bullet\beta\,]}\;\left\{\;A\rightarrow\beta\right.\tag{2.3}$$

This rule takes care of introducing active items; each rule in the grammar is added as an active item spanning $i-i$ for any possible input position $0\leq i\leq n$.

Earley-style top-down parsing

The basic algorithm is very crude, it predicts all possible inference rules on each possible position; if the grammar is large, the chart will become full of useless items.

Earley (1970) introduced a parsing algorithm, where the parse items are augmented with a *lookahead* of a number of input tokens. The algorithm with no lookahead can be simplified to a parsing system using four inference rules, two of which are the Complete and Scan rules from above. Earley prediction works in a top-down fashion; a grammar rule is predicted only when there is an item looking for the rule's left-hand side.

Combine and Scan remain as the inference rules 2.1 and 2.2.

Predict

$$\frac{[\, i - j\, ;\, C \to \gamma \bullet A\, \alpha\,]}{[\, j - j\, ;\, A \to\ \bullet \beta\,]} \left\{\ A \to \beta \right. \tag{2.4}$$

If there is an item looking for an A and ending in position j, and there is a grammar rule for A, add that rule as an active item spanning the positions $j - j$.

Initial prediction

$$\frac{}{[\, 0 - 0\, ;\, S \to\ \bullet \alpha\,]} \left\{\ S \to \alpha \right. \tag{2.5}$$

Now prediction also needs an active item to be triggered, so we need some way of starting the inference process. This is done by adding an active item for each rule of the starting category S, spanning the positions $0 - 0$.

Kilbury-style bottom-up parsing

Kilbury (1985) did a variant of Earley's algorithm, where the prediction was changed from looking top-down to bottom-up. Kilbury's algorithm is also called *left-corner* parsing in the literature (see e.g. Carroll, 2003). The basic idea is that we predict a grammar rule only when the rule looks for a category which is already found.

Combine and Scan remain as the inference rules 2.1 and 2.2.

Predict+Combine

$$\frac{[\, i - j\, ;\, B\,]}{[\, i - j\, ;\, A \to B \bullet \beta\,]} \left\{\ A \to B\, \beta \right. \tag{2.6}$$

If there is a passive item for B spanning $i - j$, and there is a rule looking for B, then we can add the rule as an active item. And since B is already found, we can apply the Combine rule immediately to move the dot forward one step.

Predict+Scan

$$\frac{}{[\, i - j\, ;\, A \to w_j \bullet \beta\,]} \left\{ \begin{array}{l} A \to w_j\ \beta \\ j = i + 1 \end{array} \right. \tag{2.7}$$

If the rule looks for a terminal, which happens to span $i - j$, then we can add that rule as an item where that terminal is found.

Note that this algorithm does not work for grammars with ϵ-rules; there is no way an empty rule can be predicted. There are two possible solutions to this; *i*) either convert the grammar to an equivalent grammar without ϵ-rules; or *ii*) add extra inference rules to handle ϵ-rules. We do not dwell further upon this issue, since this thesis is not about context-free parsing anyway.

Further modifications of the algorithms

There are several ways these basic algorithms can be optimized; e.g. by adding (top-down or bottom-up) *filtering* to the predictions. For the simple case of grammars without ϵ-rules, the *left-corner* relation is defined as,

$$X \rhd Y \quad \equiv \quad X \to Y\ \alpha$$

and the reflexive and transitive closure $\rhd^*$ is used to filter out predictions that do not match the input string or the starting category. The relation $\rhd^*$ is also known as the *first set* in LR parsing algorithms (see e.g. Aho et al., 1986).

More information about filtering and other optimizations to chart parsing algorithms can be found in e.g. Wirén (1992), Sikkel (1997b, 1998) or Nederhof and Satta (2004).

Soundness and completeness of the basic algorithm

To prove correctness we first specify the set of valid items to contain all items $[\, i-j\,;\ A \to \alpha \bullet \beta\,]$ such that $\alpha \Rightarrow^* w_{i+1} \dots w_j$. If the item is passive, then $A \Rightarrow \alpha$ and the interpretation is equivalent to $A \Rightarrow^* w_{i+1} \dots w_j$.

Soundness is easy to show, since the inference rules are quite intuitive.

Lemma 2.2. *The inference rules 2.1–2.3 are sound.*

Proof. For each inference rule we have to prove that the consequent is valid whenever the antecedents are valid;

Predict The item $[i-i\,;\ A \to \bullet\beta]$ is trivially valid, since $\epsilon \Rightarrow^* w_{i+1} \dots w_i = \epsilon$;

Scan The consequent $[i-k\,;\ A \to \alpha\ w_k \bullet \beta]$ is valid if $\alpha w_k \Rightarrow^* w_{i+1} \dots w_j w_k$; but this is equivalent to $\alpha \Rightarrow^* w_{i+1} \dots w_j$ which is true since the antecedent $[\, i-j\,;\ A \to \alpha \bullet w_{j+1}\ \beta\,]$ is valid;

Combine The consequent $[\, i-k\,;\ A \to \alpha\ B \bullet \beta\,]$ is valid if $\alpha\ B \Rightarrow^* w_{i+1} \dots w_k$; but since the first antecedent says that $\alpha \Rightarrow^* w_{i+1} \dots w_j$, and the second antecedent says that $B \Rightarrow^* w_{j+1} \dots w_k$, we get that,

$$\alpha\ B \Rightarrow^* w_{i+1} \dots w_j w_{j+1} \dots w_k = w_{i+1} \dots w_k$$

□

To prove completeness it is enough to give a *deduction length function*, assigning a natural number *rank* to each valid item. We define the function as,

$$d([\, i-k\,;\ A \to \alpha \bullet \beta\,]) \quad = \quad \min\{\, \mu + |i-k| \mid \alpha \Rightarrow^{\mu} w_{i+1} \dots w_k \,\}$$

where μ is the number of deduction steps. We take the minimum in case there are different ways to recognize an item. Completeness follows directly from the following lemma, by induction on the rank of valid items.

Lemma 2.3. *Each valid item* $[\, i - k\, ;\, A \rightarrow \alpha \bullet \beta\,]$ *with rank* d*, is a consequence of some inference rule, where the ranks of all antecedent items are less than* d*.*

Proof. There are three possibilities for the item; α can be empty, or it can either end with a terminal or a category;

- If $\alpha = \epsilon$, then $i = j$ and the item is inferred by prediction;
- If $\alpha = \alpha'\ w_k$, then the only way the item can be inferred is by scanning the item $[i - j\, ;\, A \rightarrow \alpha' \bullet w_k\ \beta]$ for $j = k - 1$. Note that the antecedent item has smaller rank since $|i - j| < |i - k|$; and the derivations $\alpha \Rightarrow^* w_{i+1} \ldots w_k$ and $\alpha' \Rightarrow^* w_{i+1} \ldots w_j$ have the same deduction lengths;
- If $\alpha = \alpha'\ B$, then $\alpha' \Rightarrow^{\mu_1} w_{i+1} \ldots w_j$ and $B \Rightarrow \gamma \Rightarrow^{\mu_2} w_{j+1} \ldots w_k$ for some $i \leq j \leq k$ and some $B \rightarrow \gamma$. The item can only be inferred by completion of the items $[\, i - j\, ;\, A \rightarrow \alpha' \bullet B\ \beta\,]$ and $[\, j - k\, ;\, B\,]$. The rank of the consequent is $d = \mu_1 + \mu_2 + 1 + |i - k|$; and the ranks of the antecedents are $\mu_1 + |i - j|$ and $\mu_2 + |j - k|$ respectively, which are both less than d.

□

2.2.3 Possible implementations

Here are two examples of how to implement the deduction engine. If it is possible to associate each parse item with a natural number *rank* such that all antecedents always are less than the consequent in an inference rule,[1] then there is a very simple implementation of a parsing system. This implementation is a generalization of the ideas of the CKY parsing algorithm (Kasami, 1965; Younger, 1967), where we use a parse array indexed by starting and ending positions in the input string.

Algorithm 2.4 (generalized CKY).

First create a parse array indexed by the ranks of the parse items, initialized with empty sets.

Then loop through each possible rank, and each possible item having that rank, adding the item to the parse matrix if some inference rule holds.

▲

Often though, there is no immediate way of associating ranks to items. In that case, we have to implement an inference engine for the given parsing system. Apart from the chart containing all items found so far, we also need an *agenda* containing the items that have not yet processed.

[1]Note that the rank may only refer to the item, not e.g. the deduction length as is done when proving completeness.

Algorithm 2.5 (agenda-driven chart parsing).

Initialize the chart to the empty set, and the agenda to all items generated by inference rules without antecedents. Then repeat the following until the agenda is empty:

- Remove one item from the agenda and add it to the chart. Apply any inference rule for which *i*) the item matches one of the antecedents, and *ii*) there are items in the chart matching the rest of the antecedents. Add all possible consequences not already in the chart to the agenda.

▲

The two algorithms can sometimes be combined; e.g. Earley (1970) uses *item sets*, one for each input token. The item sets can be created one at the time (using a generalized CKY engine), but each set is created using the agenda-driven method.

The reader is referred to e.g. Shieber et al. (1995) for more information about actual implementations.

2.2.4 Space and time complexity

The space complexity of the algorithm is (the size of the chart) times (the size of one item). For the algorithms discussed in this thesis, the size of items does not depend on the length of the input, which means that the space complexity is in the order of the number of items in the chart.

The time complexity is (the size of the chart) times (the time to infer one item). To calculate the time to infer one item, we can inspect the inference rules. The time complexity for an inference rule depends on the number of ways to instantiate the metavariables in the rule, assuming that the consequent is known. There are $\mathcal{O}(\prod |x_i|)$ possibilities for instantiating a rule, where x_i is any metavariable occurring in an antecedent or a side condition, but not in the consequent since all metavariables in the consequent are known. Here we use $|x_i|$ for the total range of a variable; e.g. if $x_i \in \mathbb{N}_n$, then $|x_i| = \mathcal{O}(n)$. Often the variables range over positions in the input string w, in which case $|x_i| = \mathcal{O}(|w|)$; or pairs of positions (representing substrings of the input), in which case $|x_i| = \mathcal{O}(|w|^2)$.

Example 2.6.

For the context-free algorithms in section 2.2.2, we get the following space and time complexities. We are only interested in complexity in the length n of the input; therefore the only variables we need to consider are those that depend on n.

An item is of the form $[\, i - j\, ;\ A \rightarrow \alpha \bullet \beta \,]$, where $0 \leq i \leq j \leq n$ and $A \rightarrow \alpha\beta$ is a grammar rule. The only variables that depend on n are i and j; therefore the space complexity is $\mathcal{O}(n^2)$.

The most complex inference rule is the Combine rule 2.1;

$$\frac{[\, i - j \, ; \; A \rightarrow \alpha \bullet B \, \beta \,] \quad [\, j - k \, ; \; B \,]}{[\, i - k \, ; \; A \rightarrow \alpha \, B \bullet \beta \,]}$$

There is only one input-dependent variable (j) which does not occur in the consequent; so the time complexity for the Combine rule is $\mathcal{O}(n)$. Since this rule is the most complex, the total time complexity of the algorithm becomes $\mathcal{O}(n^3)$.

▲

That the argument for time complexity is correct is obvious for parsing systems which can be implemented by the generalized CKY algorithm 2.4; the algorithm loops through all possible parse items (the size of the chart), and for each item it tries all possible antecedents (the time to infer one item). For the dynamic agenda-driven algorithm 2.5, the argument is a bit more complicated, but amounts to the same result (provided that there are efficient lookups for items in the chart).

2.3 Grammatical Framework

This section describes the GRAMMATICAL FRAMEWORK (GF). It is an adaption of the descriptions in Ranta (2004a,b).

A GF grammar consists of a number of *judgements*, divided into three kinds of *modules*: abstract, concrete and resource modules.

2.3.1 Abstract syntax: dependent type theory

The abstract syntax of GF is based on type theory, or to be more specific, on Martin-Löf's intuitionistic type theory (Martin-Löf, 1984). An abstract grammar is defined by giving a number of abstract judgements, which can be of the following forms:

cat $C\,[\Gamma]$	C is a category depending on the context Γ
fun $f : A$	f is a function of type A
data $C = f_1 \mid \ldots \mid f_n$	C has the constructors $f_1, \ldots, f_n$
def $a = b$	a is defined as b

Categories and types

A type declaration $a : A$ says that a is an object of type A, and presupposes that A is a type according to the type rules. A context Γ is a sequence of variable declarations $x_1 : A_1, \ldots, x_n : A_n$, where each A_i is a type whenever $x_1 : A_1, \ldots, x_{i-1} : A_{i-1}$. The are two rules for forming types, also shown in figure 2.1;

Basic type formation

$$\frac{\textbf{cat}\; C\,[x_1 : A_1,\, \ldots,\, x_n : A_n] \qquad a_1 : A_1 \quad \ldots \quad a_n : A_n[x_1/a_1,\, \ldots,\, x_{n-1}/a_{n-1}]}{C\, a_1 \ldots a_n \ \mathsf{Type}}$$

Function type formation

$$\frac{A \ \mathsf{Type} \quad B \ \mathsf{Type}\ [x : A]}{(x : A) \to B \ \mathsf{Type}}$$

Basic object formation

$$\frac{\textbf{fun}\; f : A}{f : A}$$

Function application and abstraction

$$\frac{f : (x : A) \to B \quad a : A}{f\, a : B[x/a]} \qquad \frac{b : B\,[x : A]}{\lambda x.\, b : (x : A) \to B}$$

β and η conversion

$$\frac{}{(\lambda x.\, b)\, a = b[x/a]} \qquad \frac{c : (x : A) \to B}{c = \lambda x.\, (c\, x)}$$

Figure 2.1: Rules for types and objects in abstract syntax.

- The category definition

$$\textbf{cat}\; C\,[x_1 : A_1,\, x_2 : A_2,\, \ldots,\, x_n : A_n]$$

  says that $C\, a_1 \ldots a_n$ is a type, whenever $a_1 : A_1$, $a_2 : A_2[x_1/a_1]$, $\ldots$, $a_n : A_n[x_1/a_1,\, \ldots,\, x_{n-1}/a_{n-1}]$.

- There is also a rule that creates the function type $(x : A) \to B$, if A is a type and if B is a type whenever $x : A$.

As a syntactic sugar we write $A \to B$ for the type $(x : A) \to B$ whenever B does not depend on x.

Functions and objects

There are three rules for forming objects, also shown in figure 2.1;

- The function type definition **fun** $f : A$ presupposes that A is a type, and says that f is an object of type A, or $f : A$.
- If we have a function $f : (x : A) \to B$, and an object $a : A$, we can apply the function to the object as $f\,a : B[x/a]$.
- If we have an object $b : B$ whenever $x : A$, we can form the function $\lambda x.\, b : (x : A) \to B$.

Higher-order functions and dependent types

Given a function,

$$f \quad : \quad (x_1 : B_1) \to \cdots \to (x_\delta : B_\delta) \to A$$

where A is not not a function type, we can informally define the notions of higher-order functions and dependent types;

- f is a *higher-order* function if any of the argument types B_i is a function type;
- A type B_i (or A) is *dependent* if any of the variables $x_1, \ldots, x_{i-1}$ occur in B_i (or any of $x_1, \ldots, x_\delta$ occur in A).

If a function is not higher-order and does not have dependent types, it is said to have a *context-free backbone*; this will be further discussed in section 2.3.2.

Example 2.7.

The function,

$$f \quad : \quad (x : A \to B) \to (y : C\ x) \to D$$

is higher-order since the first argument is a function, and has dependent types since $C\ x$ depends on x.

▲

Normal forms

The type of any defined function **fun** f has the form

$$(x_1 : A_1) \to \cdots \to (x_\delta : A_\delta) \to A$$

where A_i are the *argument types* and $A = C\,t_1 \ldots t_n$ is the *value type* of f. The category C is the *value category* of the function. The *full application* of f has the form $f\,a_1 \ldots a_\delta$ with type $A[x_1/a_1, \ldots, x_\delta/a_\delta]$.

A term is in $\beta\eta$-normal form if it is of the form $\lambda z_1 \to \cdots \lambda z_\delta \to b$, where b is a variable or an application of a constant, and all arguments of the application are in $\beta\eta$-normal form. The β and η conversion rules in figure 2.1 can be used to bring any well-typed term into this form. Note that a function f alone is in normal form only if its type is a basic type; in general the normal form of f is of the form $\lambda z_1 \to \cdots \to \lambda z_\delta \to f\,z_1 \ldots z_\delta$.

Curried vs. uncurried functions

We often write function applications in uncurried form, $f(a_1, \ldots, a_\delta)$. When we do this we also write the function type as the type of an uncurried function.

$$\begin{array}{rcl} f : A_1 \times \cdots \times A_\delta \to A & \equiv & f : A_1 \to \cdots \to A_\delta \to A \\ f(a_1, \ldots, a_\delta) & \equiv & f\, a_1 \ldots a_\delta \end{array}$$

The natural restriction is that A is a base type.

Data constructors and function definitions

There are some extra rules for declaring data constructors and giving function definitions. We do not go into details about these subjects, since they will not be further explored in this thesis.

CONSTRUCTOR DECLARATIONS

$$\textsf{\textbf{data}}\, C \quad = \quad f_1 \mid \ldots \mid f_n$$

presuppose that C is a category and that $f_1, \ldots, f_n$ are **fun** functions, all with value types formed by C. The judgement says that each f_i is a *constructor* that can be used in patterns in function definitions.

FUNCTION DEFINITIONS

$$\textsf{\textbf{def}}\, f\; p_1 \ldots p_\delta = t$$

presuppose that f is a **fun** function, t is an object of the value type of f, and each p_i is a *pattern* of the corresponding argument type of f. A pattern is a term formed from variables and constructors only. There is a computation rule, saying that an object $f\, a_1 \ldots a_\delta$ is equal to $t[\sigma]$, whenever each $a_i = p_i[\sigma]$, where σ is a substitution, for the *first* matching function definition. This implies that function definitions are ordered.

Functions that are neither constructors nor defined implicitly are *primitive notions.* The lexical rules of GF make no distinction between constructors, defined functions and primitive notions.

2.3.2 The context-free backbone

Most chapter of this thesis only considers a restricted variant of the abstract syntax, called the *context-free backbone.* It is not until chapter 6 that we return to higher-order functions and dependent types.

Definition 2.8 (context-free backbone). A basic function f has a context-free backbone if it has the type

$$f \quad : \quad C_1 \times \cdots \times C_\delta \to C$$

where C and all C_i are categories, without dependencies.

A grammar has a context-free backbone if all basic functions have. Another way of saying this is that there are no dependent types and no lambda abstractions in the grammar, i.e. all category declarations have empty context and there are no higher-order functions. By *context-free* GF we mean all possible GF grammars with context-free backbone.

Note that the notions of *basic types* and *categories* coincide when we talk about grammars with context-free backbones. In these cases we use the notion category, to distinguish from the linearization types of the concrete syntax. In all chapters except for chapter 6, we only talk about grammars with a context-free backbone.

The definition of $t : A$ in figure 2.1 becomes very simple for grammars with context-free backbone;

$$f(t_1, \ldots, t_\delta) : C \quad \textbf{iff} \quad t_1 : C_1, \ldots, t_\delta : C_\delta$$

whenever f has the definition given above.

Example 2.9.

We repeat our main example from section 1.3.5; a simple grammar for a fragment of English sentences. It consists of the context-free categories S, NP, VP, D, N and V (standing for Sentence, Noun Phrase, Verb Phrase, Determiner, Noun and Verb respectively), and has the following functions;

$$\begin{aligned}
s_p &\ : \ \mathsf{NP} \times \mathsf{VP} \to \mathsf{S} \\
np_d &\ : \ \mathsf{D} \times \mathsf{N} \to \mathsf{NP} \\
np_p &\ : \ \mathsf{N} \to \mathsf{NP} \\
vp_t &\ : \ \mathsf{V} \times \mathsf{NP} \to \mathsf{VP} \\
d_a, d_m &\ : \ \mathsf{D} \\
n_c, n_f &\ : \ \mathsf{N} \\
v_e &\ : \ \mathsf{V}
\end{aligned}$$

The predication function s_p forms a sentence out of a noun phrase and a verb phrase. There are two ways of forming noun phrases; either by a determiner and a noun ('*a lion*', '*many lions*'), or just a plural noun ('*lions*'). We assume that all verbs are transitive, so we only have the transitive verb phrase forming function vp_t. The determiners d_a, d_m are singular and plural indefinites ('*a*' and '*many*'); n_c, n_f are the nouns '*lion*' and '*fish*'; and v_e is the verb '*eat*'. ▲

Stripping off dependencies

Given an arbitrary GF grammar G without higher-order functions, we can strip off all dependencies from that grammar to get the context-free backbone of G. This is possible since each first-order function type can be written on the form

$$(x_1 : A_1) \to \cdots \to (x_\delta : A_\delta) \quad \to \quad C\, t_1 \ldots t_n$$

where each A_i is of the form $C_i\, t_{i,1} \ldots t_{i,n_i}$. Each typing of the given form is then translated to $C_1 \times \cdots \times C_\delta \to C$.

The resulting grammar will be *over-generating*, meaning that all type-correct terms will still be accepted by the context-free backbone, but terms that are not type-correct might also be accepted. Another way of saying this is that the translation is *complete* (i.e. all correct terms are still accepted), but not *sound* (i.e. incorrect terms are also accepted).

Higher-order functions

If there are higher-order functions in a grammar, we can still apply the same transformation. We only have to note that the argument types A_i above can be functions themselves. We then apply the transformation recursively on the argument types. However, the resulting grammar will not have a context-free backbone. Higher-order functions will remain higher-order, even when all dependencies are stripped off. The final function types will be of the form $A_1 \times \cdots \times A_\delta \to C$, where each A_i is also of the same form.

2.3.3 Concrete syntax

The concrete syntax of an abstract grammar is specified by giving a number of concrete judgements of the following three forms:

lincat $C = L$	C has the linearization type L
lin $f\, x_1 \ldots x_\delta = t$	f has the linearization function $\lambda x_1 \ldots x_\delta .\, t$
lindef $C\, x = t$	C has the default linearization $\lambda x.\, t$

Strings and tokens

The type of strings, Str, consists of sequences of *tokens*, where a token is an abstract entity. The only thing we need to know is that the set of tokens is finite and written in this thesis as Σ. In our examples, tokens are words, but they could be e.g. morphological analyses of input words.

There is one operation on strings, *concatenation* of two strings. In this thesis we write $s_1 \cdot s_2$ for the concatenation of two strings, or often even $s_1\ s_2$ when no confusion can arise. The empty string is written ϵ.

In GF it is also possible to concatenate tokens, called the *agglutination* $t_1 + t_2$ of two tokens which is also a token. This presupposes that tokens are strings, or more generally, that the set of tokens forms a monoid. The type system of GF ensures that the set Σ of tokens that is used by a specific grammar is still finite. Since agglutination can be eliminated, we will make no use of agglutination in this thesis; so we simply assume that there is a finite set of tokens.

Linearization types

A *linearization type* can be of the following forms,[2]

- The type of strings, Str, is a linearization type;
- If $T_1, \ldots, T_n$ are linearization types or parameter types, and at least one of them is a linearization type, then $\{\, r_1 : T_1 \,;\, \ldots \,;\, r_n : T_n \,\}$ is a linearization type;
- If T is a linearization type and P is a parameter type, then $P \Rightarrow T$ is a linearization type.

This means that a linearization type contains the type Str somewhere. Parameter types are defined in section 2.3.4; for now it is enough to know that they are always finite.

Each category C defined in the abstract grammar, must be given a corresponding linearization type C°, which is done by the judgement **lincat** $C = L$. For context-free backbones, the judgement simply says that $C^\circ = L$. For arbitrary types, linearization types are defined inductively as follows,[3]

$$\begin{aligned} (C\ a_1 \ldots a_n)^\circ &= L, \text{ if } \textbf{lincat}\, C = L \\ ((x_1 : A_1) \to \cdots \to (x_n : A_n) \to A)^\circ &= \mathsf{Str}^n \times A^\circ \end{aligned}$$

The second line in the definition is only used when the grammar has higher-order functions. In that case the definition specifies how a functional argument is linearized. This will be discussed further in section 6.1.

Example 2.10.

The example grammar has the following linearization types for English, where Num is a parameter type containing parameters for singular and plural;

$$\begin{aligned} \mathsf{S}^\circ &= \{\, s : \mathsf{Str} \,\} \\ \mathsf{D}^\circ, \mathsf{NP}^\circ &= \{\, s : \mathsf{Str}\,;\ n : \mathsf{Num} \,\} \\ \mathsf{N}^\circ, \mathsf{V}^\circ, \mathsf{VP}^\circ &= \{\, s : \mathsf{Num} \Rightarrow \mathsf{Str} \,\} \end{aligned}$$

[2]The definition we use here is more general than the one used by Ranta (2004a), which includes some implementation-specific restrictions.

[3]Recall that a tuple like $\mathsf{Str}^n \times A^\circ$ is just syntactic sugar for a record.

The reason for these linearization types is that nouns are inflected by determiners and that verbs are inflected by the subject noun phrase. Other languages might have other linearization types; e.g. in Swedish and German, there is an extra inflection parameter for gender, making the linearization types more complicated.

▲

Linearization functions

To each function typing in the abstract syntax,

$$\textbf{fun}\, f \quad : \quad (x_1 : A_1) \to \cdots \to (x_\delta : A_\delta) \to A$$

a corresponding *linearization rule* should be specified, which is done by a judgement of the form

$$\textbf{lin}\, f\, x_1 \ldots x_\delta \quad = \quad \phi$$

This presupposes that $\phi : A^\circ$ whenever $x_i : A_i^\circ$ for $1 \leq i \leq \delta$. The judgement defines a linearization function $f^\circ = \lambda x_1 \ldots x_\delta .\, \phi$.

Evaluating linearizations

The linearization of a compound term $t = f\, t_1 \ldots t_\delta$ is defined as

$$\begin{aligned} [\![f\, t_1 \ldots t_\delta]\!] &= f^\circ\, [\![t_1]\!] \ldots [\![t_\delta]\!] \\ &= \phi[x_1/[\![t_1]\!], \ldots, x_\delta/[\![t_\delta]\!]] \end{aligned}$$

whenever the linearization rule is specified as $\textbf{lin}\, f\, x_1 \ldots x_\delta = \phi$.

Default linearizations

The third possible judgement in a concrete module is a default linearization **lindef** $C\, x = \phi$. Each basic category C can have a default linearization, which is a function from strings to C°. The default linearization is only used when linearizing bound variables in higher-order functions. This is discussed further in section 6.1.

2.3.4 Resource syntax

The concrete syntax of GF has developed into a functional programming language, with data type definitions and local and global function definitions. In GF, data types are called *parameter types* and global functions are called *operations*, and they are specified by the following two judgements:

param $P = C_1\Gamma_1 \mid \ldots \mid C_n\Gamma_n$	P is a parameter type with constructors $C_1, \ldots, C_n$
oper $h : T = t$	h is an operation of type T defined as t

The parameter contexts Γ_i are sequences of parameter types $P_{i,1} \ldots P_{i,n_i}$. Both these judgements specify *resources* which can be used by concrete linearization types and linearization rules.

Parameter types

A *parameter type* can be of the following forms,

- If there is a parameter type declaration for P, then P is a parameter type;
- If $P_1, \ldots, P_n$ are parameter types, then the record type,

  $$\{\, r_1 : P_1 \,;\, \ldots \,;\, r_n : P_n \,\}$$

  is a parameter type.

A parameter type declaration is of the form,

$$\textbf{param}\ P \quad = \quad C_1\,\Gamma_1 \mid \ldots \mid C_n\,\Gamma_n$$

and defines a series of constructors $C_1, \ldots, C_n$ which can be used as functions from their parameter contexts to P. I.e. if $\Gamma_i = P_{i,1} \ldots P_{i,n_i}$, then the constructor C_i is a function,

$$C_i \quad : \quad P_{i,1} \to \cdots \to P_{i,n_i} \to P$$

Parameter type declarations may not be recursive; neither direct, indirect nor mutually. This ensures that every parameter type P is finite, and we can form the set of all *parameter values* of type P,

$$V_P \quad = \quad \{\, 1_P, 2_P, \ldots, n_P \,\}$$

Example 2.11.

For the simple English grammar we only need to declare one parameter type for number,

$$\textbf{param}\ \mathsf{Num} \quad = \quad \mathsf{Sg} \mid \mathsf{Pl}$$

but for more complicated grammars we also need a Person parameter type, for distinguishing first, second and third person,

$$\textbf{param}\ \mathsf{Person} \quad = \quad \mathsf{First} \mid \mathsf{Second} \mid \mathsf{Third}$$

Other languages might need other parameter types, e.g. German whose nouns and verbs also are inflected by gender and case,

$$\begin{aligned}\textbf{param}\ \mathsf{Gen} &= \mathsf{Masc} \mid \mathsf{Fem} \mid \mathsf{Neu}\\ \textbf{param}\ \mathsf{Case} &= \mathsf{Nom} \mid \mathsf{Acc} \mid \mathsf{Det} \mid \mathsf{Gen}\end{aligned}$$

If there are several inflection parameters which often come together, it is possible to define a combined parameter type,

$$\textbf{param}\ \mathsf{Infl} = \mathsf{Infl}(\mathsf{Num}, \mathsf{Gen}, \mathsf{Case})$$

This type contains $2 \times 3 \times 4 = 24$ different parameters. ▲

Operations

An *operation* is a global definition of a helper function that can be used in linearizations. An operation is defined by judgements of the form

$$\textbf{oper}\ h : T = t$$

where T is any type (in the sense of concrete syntax) and $t : T$.

The type system of concrete syntax consists of linearization types augmented with function types. A simple example of an operation definition is ss, creating a record from a string,

$$\textbf{oper}\ ss : \mathsf{Str} \rightarrow \{\, s : \mathsf{Str}\,\} = \lambda x.\{\, s = x\,\}$$

There is also a designated type Type for describing linearization types. This designated type is needed to be able to define functions that can be applied on different kinds of types. An example is a generalized version of the previous operation, creating a record from an object of any type,

$$\textbf{oper}\ rr : (a : \mathsf{Type}) \rightarrow a \rightarrow \{\, r : a\,\} = \lambda a\, x.\{\, r = x\,\}$$

Note that we use dependent types in this definition.

Grammar composition

Any GF grammar with a context-free backbone can be transformed into a resource module by the following simple translation:

- Each category C with its linearization type C° is transformed into the operation

$$\textbf{oper}\ C : \mathsf{Type} = C^\circ$$

- Each basic function $f : A$ with its linearization function f° is transformed into the operation

$$\textbf{oper}\ f : A \quad = \quad f^\circ$$

This resource module can then be used by another GF grammar, giving a way of performing grammar composition as mentioned in sections 1.3.1 and 1.3.4.

2.3.5 Linearization terms

The possible linearization terms in concrete syntax is defined inductively as in figure 2.2. Most of the definitions are straightforward. The only thing needing further explanation is the definition for table formation.

Table formation

To explain the rule for forming tables, we have to give a definition of *patterns*. Informally, a pattern is an incomplete term, where variables can occur in place of subterms. A pattern p matches a term t if there is some substitution σ such that $t = p[\sigma]$, i.e. all variables in the pattern can be instantiated to get an equivalent term. A sequence of patterns $p_1, \ldots, p_n$ is *exhaustive* if every possible term is matched by some pattern. We write $t : T[p : P]$ if $t : T$ whenever all variables in p is assumed to be of type such that $p : P$.

Example 2.12.

The linearization terms of the example grammar in section 1.3.5 are repeated here.

$$\begin{aligned}
s_p^\circ(x, y) &= \{\, s = x.s \cdot y.s \,!\, x.n \,\} \\
np_d^\circ(x, y) &= \{\, s = x.s \cdot y.s \,!\, x.n \,;\ n = x.n \,\} \\
np_p^\circ(x) &= \{\, s = x.s \,!\, \mathsf{Pl} \,;\ n = \mathsf{Pl} \,\} \\
vp_t^\circ(x, y) &= \{\, s = [\, z \Rightarrow x.s \,!\, z \cdot y.s \,] \,\} \\
d_a^\circ &= \{\, s = \text{‘}a\text{’} \,;\ n = \mathsf{Sg} \,\} \\
d_m^\circ &= \{\, s = \text{‘}many\text{’} \,;\ n = \mathsf{Pl} \,\} \\
n_c^\circ &= \{\, s = [\, \mathsf{Sg} \Rightarrow \text{‘}lion\text{’} \,;\ \mathsf{Pl} \Rightarrow \text{‘}lions\text{’} \,] \,\} \\
n_f^\circ &= \{\, s = [\, \_ \Rightarrow \text{‘}fish\text{’} \,] \,\} \\
v_e^\circ &= \{\, s = [\, \mathsf{Sg} \Rightarrow \text{‘}eats\text{’} \,;\ \mathsf{Pl} \Rightarrow \text{‘}eat\text{’} \,] \,\}
\end{aligned}$$

The full GF grammar, including the abstract syntax and the linearization types, is shown in figure 2.3.

▲

Strings

$$\text{`}\textit{string}\text{'} : \mathsf{Str} \qquad \epsilon : \mathsf{Str} \qquad \frac{s,\, t : \mathsf{Str}}{s \,\cdot\, t : \mathsf{Str}}$$

Parameter constructors

$$\frac{\textbf{param}\ P = \ldots \mid C\, P_1 \ldots P_n \mid \ldots}{C : P_1 \to \cdots \to P_n \to P}$$

Functions

$$\frac{b : B[x : A]}{\lambda x.\, b : A \to B} \qquad \frac{f : A \to B \quad a : A}{f\, a : B}$$

Records

$$\frac{t_1 : T_1 \quad \ldots \quad t_n : T_n}{\{\, r_1 = t_1\, ;\, \ldots\, ;\, r_n = t_n \,\} : \{\, r_1 : T_1\, ;\, \ldots\, ;\, r_n : T_n \,\}}$$

$$\frac{c : \{\, \ldots\, ;\, r : T\, ;\, \ldots \}}{c.r : T}$$

Tables

$$\frac{t_1 : T[p_1 : P] \quad \ldots \quad t_n : T[p_n : P]}{[\, p_1 \Rightarrow t_1\, ;\, \ldots\, ;\, p_n \Rightarrow t_n \,] : P \Rightarrow T} \left\{ \begin{array}{l} p_1, \ldots, p_n \\ \text{exhaustive} \end{array} \right.$$

$$\frac{c : P \Rightarrow T \quad p : P}{c\, !\, p : T}$$

Local definitions

$$\frac{t : T \quad e : E[x : T]}{(\textbf{let}\ x : T = t\ \textbf{in}\ e) : E}$$

Global definitions

$$\frac{\textbf{oper}\ h : T = t}{h : T}$$

Figure 2.2: The types and objects in concrete syntax.

Categories

$$\textbf{cat}\ \mathsf{S},\ \mathsf{NP},\ \mathsf{VP},\ \mathsf{D},\ \mathsf{N},\ \mathsf{V}$$

Parameter types

$$\textbf{param}\ \mathsf{Num} \quad = \quad \mathsf{Sg} \mid \mathsf{Pl}$$

Linearization types

$$\begin{aligned}
\mathsf{S}^\circ &= \{\, s : \mathsf{Str} \,\} \\
\mathsf{D}^\circ,\ \mathsf{NP}^\circ &= \{\, s : \mathsf{Str}\,;\ n : \mathsf{Num} \,\} \\
\mathsf{N}^\circ,\ \mathsf{V}^\circ,\ \mathsf{VP}^\circ &= \{\, s : \mathsf{Num} \Rightarrow \mathsf{Str} \,\}
\end{aligned}$$

Functions

$$\begin{aligned}
s_p &: \mathsf{NP} \times \mathsf{VP} \to \mathsf{S} \\
np_d &: \mathsf{D} \times \mathsf{N} \to \mathsf{NP} \\
np_p &: \mathsf{N} \to \mathsf{NP} \\
vp_t &: \mathsf{V} \times \mathsf{NP} \to \mathsf{VP} \\
d_a,\ d_m &: \mathsf{D} \\
n_c,\ n_f &: \mathsf{N} \\
v_e &: \mathsf{V}
\end{aligned}$$

Linearization functions

$$\begin{aligned}
s_p^\circ(x,\ y) &= \{\, s = x.s \cdot y.s\,!\,x.n \,\} \\
np_d^\circ(x,\ y) &= \{\, s = x.s \cdot y.s\,!\,x.n\,;\ n = x.n \,\} \\
np_p^\circ(x) &= \{\, s = x.s\,!\,\mathsf{Pl}\,;\ n = \mathsf{Pl} \,\} \\
vp_t^\circ(x,\ y) &= \{\, s = [\, z \Rightarrow x.s\,!\,z \cdot y.s \,] \,\} \\
d_a^\circ &= \{\, s = \text{‘}a\text{’}\,;\ n = \mathsf{Sg} \,\} \\
d_m^\circ &= \{\, s = \text{‘}many\text{’}\,;\ n = \mathsf{Pl} \,\} \\
n_c^\circ &= \{\, s = [\, \mathsf{Sg} \Rightarrow \text{‘}lion\text{’}\,;\ \mathsf{Pl} \Rightarrow \text{‘}lions\text{’} \,] \,\} \\
n_f^\circ &= \{\, s = [\, \_ \Rightarrow \text{‘}fish\text{’} \,] \,\} \\
v_e^\circ &= \{\, s = [\, \mathsf{Sg} \Rightarrow \text{‘}eats\text{’}\,;\ \mathsf{Pl} \Rightarrow \text{‘}eat\text{’} \,] \,\}
\end{aligned}$$

Figure 2.3: Example grammar for a small fragment of English.

Computation rules

There are straightforward computation rules for string concatenation, local and global definitions, function application, record projection and table selection.

$$\begin{array}{rcll} s_1 \cdot s_2 & = & s_1 s_2 & \\ \textbf{let}\ x : T = t\ \textbf{in}\ e & = & e[x/t] & \\ h & = & t & (\textbf{oper}\ h : T = t) \\ (\lambda x.\, t)\, a & = & t[x/a] & \\ \{\ldots\,;\, r = t\,;\, \ldots\}.r & = & t & \\ [\ldots\,;\, p \Rightarrow t\,;\, \ldots]\,!\, s & = & t[p/s] & (p \text{ matches } s \text{ first}) \end{array}$$

The restriction for the last rule means that p must be the *first* pattern in the table that matches s. This together with the fact that the patterns in a table are exhaustive, ensures that table selection is a deterministic function. By $[p/s]$ we mean that we apply the unique substitution σ such that $s = p[\sigma]$.

2.3.6 The module system

Here we note some remarks about the module system in GF. Ranta (2004b) gives a more detailed description of these things.

Interface modules

There are actually four kinds of modules; the fourth being *interface modules*. An interface module is like a restricted resource module, where only the *types* of operations are declared, not the implementations, and only the *names* of parameter types are declared. This gives an analogy between *abstract* and *concrete* on the top level and *interface* and *instance* on the resource level.

Extension and inheritance

Any module can *extend* one or more modules of the same kind. The new module then inherits all definitions from the underlying module. A module can extend another (unrelated) module, and it can also be extended by any number of (unrelated) modules. In this way we can form a hierarchy of modules.

Example 2.13.

Assume that we have an abstract module Logic defining propositions and the logical connectives. We can have several concrete syntaxes for this module, LogicDan, LogicSwe and LogicLatex for linearizing to Danish, Swedish and LaTeX. Now we can extend the Logic module by another abstract module Arithm for arithmetic. The concrete syntaxes ArithmDan, ArithmSwe and ArithmLatex, can be implemented as extensions of the corresponding concrete logic modules.

This example can be further extended by extending Logic in another direction, such as Geom for geometry, together with corresponding concrete syntaxes. Then a final abstract module GeoAri can be declared as the extension of both Geom and Arithm, and the corresponding concrete syntaxes can be specified in the same way.

Finally, we can add a resource module hierarchy which is used be the concrete modules, e.g. LatexRes containing LaTeX features, ScanRes containing common features for the Scandinavian languages, together with DanRes and SweRes which are both extensions of ScanRes and contains language-specific features. The concrete module XY (for X=Logic, Arithm, Geom, GeoAri; and Y=Dan, Swe, Latex) can then make use of the resource module YRes.

▲

2.3.7 Canonical linearizations

The concrete syntax of any GF grammar can be partially evaluated to a grammar in canonical form, as shown in Ranta (2004a). In canonical form, all local and global definitions disappear, as well as function applications; furthermore, all tables are instantiated, meaning that all patterns are variable-free. Hierarchical parameters can be flattened; thus we can assume that the parameters are declared by giving a finite set Par of parameter types, each $P \in \mathsf{Par}$ being a set of parameters $p_1, \ldots, p_n$. The resulting possible linearization functions and terms are defined by the following.

Linearization functions

A linearization function for $f : B_1 \times \cdots \times B_\delta \to A$ in canonical GF is of the form,

$$f^\circ(x_1, \ldots, x_\delta) \quad = \quad \phi$$

where ϕ is a canonical linearization term.

Linearization terms

Definition 2.14 (canonical linearization). A canonical linearization term is of the following form:

- A string constant is of type Str; and a concatenation $s_1 \cdot s_2 : \mathsf{Str}$, whenever $s_1, s_2 : \mathsf{Str}$.
- A constant parameter $p : P$, whenever $p \in P$.

- A record $\{ r_1 = \phi_1 ; \ldots ; r_n = \phi_n \}$ is of type $T = \{ r_1 : T_1 ; \ldots ; r_n : T_n \}$, whenever each $\phi_i : T_i$.
- A record projection $\phi.r_i : T_i$, whenever ϕ is of the record type T above.
- A table $[p_1 \Rightarrow \phi_1 ; \ldots ; p_n \Rightarrow \phi_n]$ is of type $P \Rightarrow T$, whenever $P = \{ p_1, \ldots, p_n \}$, and each $\phi_i : T$.
- A table selection $\phi \,!\, \psi : T$, whenever $\phi : P \Rightarrow T$ and $\psi : P$.
- An argument variable $x_i : B_i^\circ$.

Example 2.15.

The example grammar in figure 2.3 is not entirely in canonical form. The linearization of vp_t contains a non-expanded table $[z \Rightarrow x.s \,!\, z \cdot y.s]$, whose canonical form is $[\mathsf{Sg} \Rightarrow x.s \,!\, \mathsf{Sg} \cdot y.s ; \mathsf{Pl} \Rightarrow x.s \,!\, \mathsf{Pl} \cdot y.s]$; and the linearization of n_f contains an anonymous table $[\_ \Rightarrow$ '*fish*' $]$, whose canonical form is $[\mathsf{Sg} \Rightarrow$ '*fish*' ; $\mathsf{Pl} \Rightarrow$ '*fish*' $]$. The full grammar in canonical form is shown in figure 2.4. ▲

Computation rules

Together with this there are computation rules for string concatenation, record projection and table selection. Since all tables are instantiated, table selection becomes as simple as record projection.

$$\begin{array}{rcl} s_1 \cdot s_2 & = & s_1 s_2 \\ \{ \ldots ; r = t ; \ldots \}.r & = & t \\ [\ldots ; p = t ; \ldots] \,!\, p & = & t \end{array}$$

2.3.8 A note on the syntax of GF grammars

There are some differences between the notation for GF used in this thesis, and the notation used in the actual implementation (GF, 2004); the main differences are shown in figure 2.5.

2.4 Generalized context-free grammar

Generalized context-free grammar (gcfg) was introduced by Pollard in the 80's as a way of formally describing head grammar (Pollard, 1984). In later work people have used gcfg as a framework for describing many other formalisms, such as linear context-free rewriting systems (Vijay-Shanker et al., 1987) and parallel multiple context-free grammar (Seki et al., 1991); and here we will use it to describe gf with a context-free backbone.

Categories

$$\textbf{cat}\ \mathsf{S},\ \mathsf{NP},\ \mathsf{VP},\ \mathsf{D},\ \mathsf{N},\ \mathsf{V}$$

Parameter types

$$\textbf{param}\ \mathsf{Num} \quad = \quad \mathsf{Sg} \mid \mathsf{Pl}$$

Linearization types

$$\begin{aligned}
\mathsf{S}^\circ &= \{\, s : \mathsf{Str} \,\} \\
\mathsf{D}^\circ,\ \mathsf{NP}^\circ &= \{\, s : \mathsf{Str}\,;\ n : \mathsf{Num} \,\} \\
\mathsf{N}^\circ,\ \mathsf{V}^\circ,\ \mathsf{VP}^\circ &= \{\, s : \mathsf{Num} \Rightarrow \mathsf{Str} \,\}
\end{aligned}$$

Functions

$$\begin{aligned}
s_p &: \mathsf{NP} \times \mathsf{VP} \to \mathsf{S} \\
np_d &: \mathsf{D} \times \mathsf{N} \to \mathsf{NP} \\
np_p &: \mathsf{N} \to \mathsf{NP} \\
vp_t &: \mathsf{V} \times \mathsf{NP} \to \mathsf{VP} \\
d_a,\ d_m &: \mathsf{D} \\
n_c,\ n_f &: \mathsf{N} \\
v_e &: \mathsf{V}
\end{aligned}$$

Linearization functions

$$\begin{aligned}
s_p^\circ(x,\ y) &= \{\, s = x.s \cdot y.s \,!\, x.n \,\} \\
np_d^\circ(x,\ y) &= \{\, s = x.s \cdot y.s \,!\, x.n\,;\ n = x.n \,\} \\
np_p^\circ(x) &= \{\, s = x.s \,!\, \mathsf{Pl}\,;\ n = \mathsf{Pl} \,\} \\
np_t^\circ(x,\ y) &= \{\, s = [\, \mathsf{Sg} \Rightarrow x.s \,!\, \mathsf{Sg} \cdot y.s\,;\ \mathsf{Pl} \Rightarrow x.s \,!\, \mathsf{Pl} \cdot y.s \,] \,\} \\
d_a^\circ &= \{\, s = \text{`}a\text{'}\,;\ n = \mathsf{Sg} \,\} \\
d_m^\circ &= \{\, s = \text{`}many\text{'}\,;\ n = \mathsf{Pl} \,\} \\
n_c^\circ &= \{\, s = [\, \mathsf{Sg} \Rightarrow \text{`}lion\text{'}\,;\ \mathsf{Pl} \Rightarrow \text{`}lions\text{'} \,] \,\} \\
n_f^\circ &= \{\, s = [\, \mathsf{Sg} \Rightarrow \text{`}fish\text{'}\,;\ \mathsf{Pl} \Rightarrow \text{`}fish\text{'} \,] \,\} \\
v_e^\circ &= \{\, s = [\, \mathsf{Sg} \Rightarrow \text{`}eats\text{'}\,;\ \mathsf{Pl} \Rightarrow \text{`}eat\text{'} \,] \,\}
\end{aligned}$$

Figure 2.4: Example grammar in canonical form.

Notion	**In this thesis**	**In the implementation**
Function type	$B_1 \times \cdots \times B_n \to A$	`B1 -> ... -> Bn -> A`
Function application	$f(\phi, \ldots, \psi)$	`f phi ... psi`
λ-abstraction	$\lambda x.\,\phi$	`\x -> phi`
String token	'*token*'	`"token"`
Concatenation	$\phi \cdot \psi$	`phi ++ psi`
Empty string	ϵ	`[]`
Table	$[\,p \Rightarrow \phi;\; q \Rightarrow \psi\,]$	`table {p => phi; q => psi}`

Figure 2.5: Notational differences between this thesis and the implementation.

There are several definitions of GCFG in the literature; Seki et al. (1991) use a definition similar to Pollard's original, while others (Weir, 1988; Becker, 1994; Chiang, 2001) more cleanly separates between abstract and concrete syntax. However, the latter definitions use the term GCFG for only the abstract part of the grammar, and the term CONTEXT-FREE REWRITING SYSTEM for the GCFG together with the concrete interpretation function. While Pollard imposed no restriction on the concrete linearization type, other definitions restrict them to be tuples of strings. In this thesis we stick to the original definition as much as possible, but separate the abstract and concrete syntax in a manner similar to the definitions of CONTEXT-FREE REWRITING SYSTEMS.

2.4.1 Abstract grammar

The abstract grammar is a tuple $(\mathcal{C}, S, \mathcal{F}, \mathcal{R})$, where $\mathcal{C}$ and $\mathcal{F}$ are finite sets of categories and function symbols respectively, $S \in \mathcal{C}$ is the starting category, and $\mathcal{R} \subseteq \mathcal{C} \times \mathcal{F} \times \mathcal{C}^*$ is a finite set of context-free syntax rules. For each function symbol $f \in \mathcal{F}$ there is an associated context-free syntax rule;

$$A \quad \to \quad f[B_1, \ldots, B_\delta]$$

The *arity* of the rule is δ, and in general we write δ_f for the arity of the rule f. The tree rewriting relation $t : A$ is defined as $f(t_1, \ldots, t_\delta) : A$ whenever $t_1 : B_1, \ldots, t_\delta : B_\delta$. We say that a tree t is *valid* (for a given category A) if $t : A$.

Example 2.16.

The abstract syntax of the example grammar (figure 2.3, 2.4), becomes as follows in GCFG format;

$$\begin{array}{rcl}
\mathsf{S} & \rightarrow & s_p[\mathsf{NP},\ \mathsf{VP}] \\
\mathsf{NP} & \rightarrow & np_d[\mathsf{D},\ \mathsf{N}] \\
\mathsf{NP} & \rightarrow & np_p[\mathsf{N}] \\
\mathsf{VP} & \rightarrow & vp_t[\mathsf{V},\ \mathsf{NP}] \\
\mathsf{D} & \rightarrow & d_a[] \\
\mathsf{D} & \rightarrow & d_m[] \\
\mathsf{N} & \rightarrow & n_c[] \\
\mathsf{N} & \rightarrow & n_f[] \\
\mathsf{V} & \rightarrow & v_e[]
\end{array}$$

▲

2.4.2 Concrete interpretation

To each category A is associated a *linearization type* A°, which is not further specified. To each function symbol f is associated a partial *linearization function* f°, taking as many arguments as the abstract syntax rule specifies.

$$f^\circ \quad \in \quad B_1^\circ \times \cdots \times B_\delta^\circ \rightarrow A^\circ$$

The linearization of a syntax tree is defined as,

$$[\![f(t_1, \ldots, t_\delta)]\!] \quad = \quad f^\circ([\![t_1]\!], \ldots, [\![t_\delta]\!])$$

if the application is defined. Note that the definition imposes no restrictions on the linearization types or the linearization functions; this is left to the actual grammar formalism. For our purposes it is enough to view a linearization type as the set of all possible linearization values. This means that the type Str of strings is equal to Σ^* (where Σ is the string alphabet). With this view we can say that a linearization type is *finite* when it is a finite set.

In section 5.2, we extend the definition to also contain *many-valued* linearization functions. Then the linearization of a tree becomes a many-valued function,

$$[\![f(t_1, \ldots, t_\delta)]\!] \quad = \quad f^\circ(\phi_1, \ldots, \phi_\delta)$$

whenever there are $\phi_1, \ldots, \phi_\delta$ such that $\phi_1 = [\![t_1]\!], \ldots, \phi_\delta = [\![t_\delta]\!]$.

2.4.3 Variable-free notation for linearizations

In some cases it is more convenient to describe the abstract syntax and the concrete linearization at the same time, without using any variable bindings.

The rule $A \to f[A_1, \ldots, A_\delta]$, with its linearization $f^\circ(x_1, \ldots, x_\delta) = \phi$, can be written as

$$A \to f[A_1, \ldots, A_\delta] := \hat{\phi}$$

where each occurrence of the variable x_i in ϕ is replaced by the term A_i in $\hat{\phi}$, representing the argument number and the argument category. This means that the basic operations of the linearization type also must have an explicit representation in $\hat{\phi}$. If more than one argument have the same category, we use superscripts to separate between them.

Example 2.17.

The following is an artificial rule in variable-free form (where the linearization types for A, B are strings):

$$A \to f[B^1, A, B^3] \quad := \quad \text{‘}a\text{’}\ B^1\ A\ \text{‘}b\text{’}\ B^3$$

This rule is another way of writing the linearization function

$$f^\circ(x, y, z) \quad = \quad \text{‘}a\text{’}\ x\ y\ \text{‘}b\text{’}\ z$$

Which in turn is a GCFG version of the context-free rule $A \to$ ‘a’ B A ‘a’ B.

▲

2.4.4 Subclasses of GCFG

When defining subclasses of GCFG, we use a notion of “a part of x”, which can be defined in terms of projection functions as follows.

- If there is a bijective function $\pi : T \to P_1 \times \cdots \times P_n$, we say that π forms a *partition* of T.
- Given a term $t : T$, we say that a projected term $p_k : P_k$ is a *part* of t if there is some partition π of T such that $p_k = \pi_k(\pi(t))$.

Note that it is important that the partition is bijective, i.e. one-to-one and onto; *i*) one-to-one ensures that it is possible to reconstruct terms from the image of the partition; and *ii*) onto ensures that there is no overlapping information in the image.

Definition 2.18. Given a GCFG rule $A \to f[B_1, \ldots, B_\delta]$ with its linearization $f(x_1, \ldots, x_\delta) = \phi$. We say that the rule is

PARALLEL OR NONLINEAR if some part of x_i is mentioned twice in ϕ, for some $1 \leq i \leq \delta$;

LINEAR if no part of x_i is mentioned twice in ϕ, for all $1 \leq i \leq \delta$;

ERASING if some part of x_i is not mentioned at all in ϕ, for some $1 \leq i \leq \delta$;

NONERASING if all parts of x_i are mentioned in ϕ, for all $1 \leq i \leq \delta$;

SUPPRESSING if x_i is not mentioned at all in ϕ, for some $1 \leq i \leq \delta$.

Note that *linear* and *parallel* are opposites of each other, as are *erasing* and *nonerasing*.

Example 2.19.

A record type $T = \{ r_1 : T_1 ; \ldots ; r_n : T_n \}$ has a natural projection function $\pi : T \rightarrow T_1 \times \cdots \times T_n$:

$$\pi(\{ r_1 = t_1 ; \ldots ; r_n = t_n \}) \quad = \quad \langle t_1, \ldots, t_n \rangle$$

▲

2.4.5 GF with a context-free backbone

GRAMMATICAL FRAMEWORK with a context-free backbone is an instance of GCFG, where the abstract GF rule,

$$f \quad : \quad B_1 \rightarrow \cdots \rightarrow B_\delta \rightarrow A$$

is just another way of writing the abstract GCFG rule,

$$A \quad \rightarrow \quad f[B_1, \ldots, B_\delta]$$

We also see that the GF definition of $t : A$ in section 2.3.2 is equivalent to the GCFG definition of $t : A$ in section 2.4.1; and that the GF definition of $[\![t]\!]$ in section 2.3.3 is exactly the same as the corresponding GCFG definition in section 2.4.2.

2.5 Parallel multiple context-free grammar

PARALLEL MULTIPLE CONTEXT-FREE GRAMMAR (PMCFG; Kasami et al., 1988; Seki et al., 1991) were introduced in the late 80's as a very expressive formalism, incorporating LINEAR CONTEXT-FREE REWRITING SYSTEMS and other mildly context-sensitive formalisms, but still with a polynomial parsing algorithm. PMCFG is an instance of GCFG, with the following restrictions on linearizations:

- Linearization types are restricted to tuples of strings:

  Each PMCFG grammar defines a *linearization arity* $d(C)$ for each category C; the linearization types can then be defined as $C^\circ = \mathsf{Str}^{d(C)}$.

- The only allowed operations in linearization functions are tuple projections and string concatenations:

  Each PMCFG linearization function is of the form

  $$f^\circ\left(\langle x_{1,1},\,\ldots,\,x_{1,d_1}\rangle\,,\,\ldots,\,\langle x_{\delta,1},\,\ldots,\,x_{\delta,d_\delta}\rangle\right) \quad = \quad \langle \alpha_1,\,\ldots,\,\alpha_d\rangle$$

  where each α_i is a sequence of variables $x_{j,k}$ or constant strings.

Since records can be seen as syntactic sugar for tuples, we can use records in this thesis without changing the definition of PMCFG. The linearization function above will then be written

$$f^\circ(x_1,\,\ldots,\,x_\delta) \quad = \quad \{\, \mathbf{1} = \hat\alpha_1\,;\,\ldots\,;\,\mathbf{d} = \hat\alpha_d\,\}$$

where each variable $x_{j,k}$ in α_i is replaced by the projection $x_j.\mathbf{k}$ in $\hat\alpha_i$.

2.5.1 Variable-free notation for PMCFG grammars

When writing a PMCFG grammar in variable-free notation, we often write the linearization record as a sequence of rows; in other words we leave out the opening and closing braces and replace semicolon by comma. With this simplification, the following rule for f,

$$\begin{array}{rcl} A & \rightarrow & f[B_1,\,\ldots,\,B_\delta] \\ f^\circ(x_1,\,\ldots,\,x_\delta) & = & \{\, r_1 = \alpha_1\,;\,\ldots\,;\,r_n = \alpha_n\,\} \end{array}$$

can equivalently be written,

$$A \rightarrow f[B_1,\,\ldots,\,B_\delta] \quad := \quad r_1 = \hat\alpha_1,\,\ldots,\,r_n = \hat\alpha_n$$

where each each occurrence of variable x_i in any α_k is replaced by B_i in $\hat\alpha_k$.

Example 2.20.

Figure 2.6 shows a PMCFG version of the example grammar in figure 2.3, recognizing the same strings. Since PMCFG linearizations cannot contain information about inflection, we have to move that information into the categories instead. ▲

2.5.2 Comparison with GF

Written in record notation, PMCFG becomes a trivial instance of context-free GF, without using tables and table selections. For the reverse direction, we see that any GF grammar with a context-free backbone fulfilling the following restrictions can be trivially converted to an equivalent PMCFG:

$$\begin{array}{rcl}
\mathsf{S} \to s_{p1}[\mathsf{NP}_1,\ \mathsf{VP}] & := & s = \mathsf{NP}_1.s \cdot \mathsf{VP}.s_1 \\
\mathsf{S} \to s_{p2}[\mathsf{NP}_2,\ \mathsf{VP}] & := & s = \mathsf{NP}_2.s \cdot \mathsf{VP}.s_2 \\
\mathsf{NP}_1 \to np_{d1}[\mathsf{D}_1,\ \mathsf{N}] & := & s = \mathsf{D}_1.s \cdot \mathsf{N}.s_1 \\
\mathsf{NP}_2 \to np_{d2}[\mathsf{D}_2,\ \mathsf{N}] & := & s = \mathsf{D}_2.s \cdot \mathsf{N}.s_2 \\
\mathsf{NP}_2 \to np_p[\mathsf{N}] & := & s = \mathsf{N}.s_2 \\
\mathsf{VP} \to vp_{t1}[\mathsf{V},\ \mathsf{NP}_1] & := & s_1 = \mathsf{V}.s_1 \cdot \mathsf{NP}_1.s,\ \ s_2 = \mathsf{V}.s_2 \cdot \mathsf{NP}_1.s \\
\mathsf{VP} \to vp_{t2}[\mathsf{V},\ \mathsf{NP}_2] & := & s_1 = \mathsf{V}.s_1 \cdot \mathsf{NP}_2.s,\ \ s_2 = \mathsf{V}.s_2 \cdot \mathsf{NP}_2.s \\
\mathsf{D}_1 \to d_a[] & := & s = \text{‘}a\text{’} \\
\mathsf{D}_2 \to d_m[] & := & s = \text{‘}many\text{’} \\
\mathsf{N} \to n_c[] & := & s_1 = \text{‘}lion\text{’},\ \ s_2 = \text{‘}lions\text{’} \\
\mathsf{N} \to n_f[] & := & s_1 = \text{‘}fish\text{’},\ \ s_2 = \text{‘}fish\text{’} \\
\mathsf{V} \to v_e[] & := & s_1 = \text{‘}eats\text{’},\ \ s_2 = \text{‘}eat\text{’}
\end{array}$$

Figure 2.6: PMCFG version of the example grammar.

- Records containing parameters are not allowed;
- All tables and all table selections must be instantiated.

The fact that GF can have nested records constitutes no problem – all nestings can be flattened. Also, an expanded table,

$$[\, p_1 \Rightarrow \phi_1\,;\ \ldots\,;\ p_n \Rightarrow \phi_n\,] \quad : \quad P \Rightarrow T$$

is equivalent to a record,

$$\{\, p_1 = \phi_1\,;\ \ldots\,;\ p_n = \phi_n\,\} \quad : \quad \{\, p_1 : T\,;\ \ldots\,;\ p_n : T\,\}$$

and an instantiated selection $\phi\,!\,p_i$ is equivalent to a record projection $\phi.p_i$.

Why GF is not obviously equivalent to PMCFG

When the GF grammar contains parameters in some record, or when some table is not instantiated, or when some table selection is not instantiated, the equivalence is not trivial.

- There is a table in the grammar which is not fully instantiated, e.g.

$$n_f^o \quad = \quad \{\, s = [\,\_ \Rightarrow \text{‘}fish\text{’}\,]\,\}$$

- There is a table selection where the selector is not an instantiated parameter, e.g.

$$s_p^\circ \quad = \quad \{\, s = x.s \,\cdot\, y.s \,!\, x.n \,\}$$

- There is a record in the grammar that contains a parameter, e.g.

$$d_m^\circ \quad = \quad \{\, s = \text{‘}\mathit{many}\text{’}\,;\; n = \mathsf{Pl} \,\}$$

The case of non-instantiated tables can be solved by compiling the grammar into canonical form (see section 2.3.7), where all tables are instantiated.

The remaining cases are discussed in chapter 3, where it is shown that GF and PMCFG are equivalent.

2.5.3 Linearity and nonerasingness

The name *parallel* MCFG (PMCFG) comes from the possibility of writing parallel grammars. If the grammar is linear as defined in 2.4.4 it is called a *linear* MCFG (LMCFG). If the grammar is also nonerasing, it is called a LINEAR CONTEXT-FREE REWRITING SYSTEM (LCFRS).

Theorem 2.21 (Seki et al., 1991). *For each erasing* PMCFG *(*LMCFG*) there is an equivalent nonerasing* PMCFG *(*LMCFG*).*

This implies that LMCFG and LCFRS are equivalent formalisms.

2.6 Representations of syntactical information

2.6.1 Syntax trees or abstract terms

A syntax tree for a GCFG grammar is also known as an abstract term. The following is a repetition of the tree rewriting relation defined in figure 2.1 (for GF) and section 2.4.1 (for GCFG).

Definition 2.22 (syntax tree). Given a GCFG grammar, the tree $t = f(\vec{t})$ is a legal *syntax tree* of category A, written $t : A$, iff $A \to f[\vec{A}]$ and $\vec{t} : \vec{A}$.[4]

Note that the definition is equivalent to the definition of the abstract syntax of GF in figure 2.1 on page 38, restricted to context-free categories. By the statement $t : A \Rightarrow \phi$ we mean both $t : A$ and that $[\![t]\!] = \phi$, and by $A \Rightarrow \phi$ we mean $t : A \Rightarrow \phi$ for some tree t.

[4]Note that we write $\vec{t} : \vec{A}$ instead of $t_1 : A_1, \ldots, t_\delta : A_\delta$, as discussed in section 2.1.1.

The set of all syntax trees in a grammar G for a category A linearizing to ϕ is defined as

$$\mathcal{T}_G(A, \phi) \quad = \quad \{\, t \mid t : A \Rightarrow \phi \,\}$$

The set of all syntax trees for a category A is written $\mathcal{T}_G(A) = \bigcup_\phi \mathcal{T}_G(A, \phi)$, and the set of all syntax trees for the grammar is $\mathcal{T}_G = \bigcup_A \mathcal{T}_G(A)$. Note that $\mathcal{T}_G$ consists of all syntax trees of any category, not just the starting category. If G is understood from the context we can safely skip the subscript.

The *size* of a tree $t = f(\vec{t})$ is equal to the number of function symbols in the tree;

$$|t| \quad = \quad 1 + |\vec{t}| \;=\; 1 + \sum_{i=1}^{\delta} |t_i|$$

Example 2.23.

If we want to list all possible trees for category NP in the example grammar, we can proceed as follows. First we see that there are only two ways to build an NP, and that is from the functions np_d and np_p; the former having 4 possibilities and the latter 2 possibilities;

$$\begin{array}{rl} np_d(d_a, n_c) & np_d(d_a, n_f) \\ np_d(d_m, n_c) & np_d(d_m, n_f) \\ np_p(n_c) & np_p(n_f) \end{array}$$

An example linearization can be,

$$np_d(d_m, n_c) : \mathsf{NP} \Rightarrow \text{‘}\textit{many lions}\text{’}$$

▲

Open and incomplete trees

In chapter 4, we will make use of *open trees*. These are trees where some nodes consist of *metavariables*, which are variables representing an as yet unknown tree of the correct category. We write metavariables as **?**.

Definition 2.24 (open tree). A tree $t : A$ is *open* if it is either a metavariable $t = \textbf{?}$; or if it is of the form $t = f(\vec{t})$, for the rule $A \to f[\vec{B}]$, where each $t_i : B_i$ is an open tree. A tree is *incomplete* or *uninstantiated* if it contains metavariables, otherwise it is called complete or instantiated.

The linearization of a metavariable **?** is the identity $[\![\textbf{?}]\!] = \textbf{?}$. In some cases we use argument variables as metavariables; then we can say that given a rule $A \to f[\vec{B}]$ with linearization function $f^\circ(\vec{x}) = \phi$, the uninstantiated tree $f(\vec{x})$ has linearization $[\![f(\vec{x})]\!] = \phi$.

When calculating the size of an open tree, we say that a metavariable has size 0; i.e. $|f(\vec{t})| = 1 + |\vec{t}|$ and $|\textbf{?}| = 0$.

2.6.2 Syntax forests or charts

If S is the starting category of a GCFG grammar, the set $\mathcal{T}(S, w)$ consists of all syntax trees linearizing to the input string w. For context-free grammars it is possible to represent $\mathcal{T}(S, w)$ by another CFG with precisely these syntax trees (up to renaming of non-terminals), generating the singleton language $\{ w \}$.[5] This new grammar is called a parse forest, and each tree in $\mathcal{T}(S, w)$ can be extracted in turn by a simple procedure.

This idea was introduced by Lang and Billot (Lang, 1974; Billot and Lang, 1989), and has been extended other formalisms (Vijay-Shanker and Weir, 1990, 1993b; Lang, 1994). In fact the idea works even when the input is a regular language (Lang, 1991), e.g. an incomplete sentence or output from a speech recognizer. The result is based on the construction of the intersection between a context-free grammar and a regular set by Bar-Hillel et al. (1964). The parse forest can be stored in polynomial space, even if it represents an exponential number of trees (or even an infinite number of trees in pathological cases).

In this section we extend the notion of parse forests to GCFG; where a forest can be seen as the abstract part of some GCFG grammar.

Definition 2.25 (item). An *item* is of the form $[\, A \rightarrow f[\vec{B}] \,;\, \phi \,;\, \vec{\psi} \,]$, where $A \rightarrow f[\vec{B}]$ is a rule in the grammar and $\phi = f^{\circ}(\vec{\psi})$.

An item $\theta = [\, A \rightarrow f[\vec{B}] \,;\, \phi \,;\, \vec{\psi} \,]$ can be viewed as an abstract GCFG rule,

$$A^{\phi} \quad \rightarrow \quad f_{\theta}[B_1^{\psi_1}, \ldots, B_{\delta}^{\psi_{\delta}}]$$

All definitions and results in this section can be reformulated to work on these kinds of abstract rules instead of items.

The set $[\, A \,;\, \phi \,]$ contains all items $[\, A \rightarrow f[\vec{B}] \,;\, \phi \,;\, \vec{\psi} \,]$ such that $\vec{B} \Rightarrow \vec{\psi}$. Note that $A \Rightarrow \phi$ is a consequence of any such item; since there are trees $\vec{t}$ such that $\vec{t} : \vec{B} \Rightarrow \vec{\psi}$, we see that $f(\vec{t}) : A$ and $[\![f(\vec{t})]\!] = f^{\circ}([\![\vec{t}]\!]) = f^{\circ}(\vec{\psi}) = \phi$; hence $f(\vec{t}) : A \Rightarrow \phi$. The reverse direction also holds; all items such that $A \Rightarrow \phi$ are contained in $[\, A \,;\, \phi \,]$. This is because the only way to build trees $t : A$ is by application of some rule, by which we get an item that is contained in $[\, A \,;\, \phi \,]$.

We say that an item $[\, A \rightarrow f[\vec{B}] \,;\, \phi \,;\, \vec{\psi} \,]$ *represents* a tree $t = f(\vec{t})$ iff $\vec{t} : \vec{B} \Rightarrow \vec{\psi}$. Note that there can be items that do not represent any tree. However, all items in $[\, A \,;\, \phi \,]$ represent at least one tree. We also say that a tree t is represented by $[\, A \,;\, \phi \,]$ iff it contains an item representing t. Finally we note that $[\, A \,;\, \phi \,]$ represents a tree t iff $t : A \Rightarrow \phi$; this is because $[\, A \,;\, \phi \,]$ contains exactly the items such that $A \Rightarrow \phi$.

[5]If w is not recognized by G, the new CFG generates the empty language.

Example 2.26.

$$\begin{aligned}[\quad & \mathsf{S} \to s_p[\mathsf{NP},\, \mathsf{VP}]\,; \\ & \{\, s = \text{‘}\textit{many lions eat fish}\text{’} \,\}\,; \\ & \{\, s = \text{‘}\textit{many lions}\text{’}\,;\ n = \mathsf{Pl} \,\}, \\ & \{\, s = [\, \mathsf{Sg} \Rightarrow \text{‘}\textit{eats fish}\text{’}\,;\ \mathsf{Pl} \Rightarrow \text{‘}\textit{eat fish}\text{’} \,]\,\} \quad]\end{aligned}$$

is an item for the example GF grammar in figure 2.4, representing the tree,

$$s_p(np_d(d_m,\, n_c),\ vp_t(v_e,\, np_p(n_f)))$$

The corresponding item for the PMCFG version of the grammar in figure 2.6 looks like,

$$\begin{aligned}[\quad & \mathsf{S} \to s_{p2}[\mathsf{NP}_2,\, \mathsf{VP}]\,; \\ & s = \text{‘}\textit{many lions eat fish}\text{’}\,; \\ & s = \text{‘}\textit{many lions}\text{’}\,; \\ & s_1 = \text{‘}\textit{eats fish}\text{’},\ s_2 = \text{‘}\textit{eat fish}\text{’} \quad]\end{aligned}$$

▲

Definition 2.27 (syntax forest, chart). A *syntax forest*, or *chart*, is a (possibly infinite) set of items.

We say that a chart $\mathcal{C}$ represents a tree $t = f(\vec{t})$ iff there is an item,

$$[\, A \to f[\vec{B}]\,;\ \phi\,;\ \vec{\psi}\,] \quad \in \quad \mathcal{C}$$

such that $[\![t]\!] = \phi$ and the subtrees $\vec{t}$ are represented by the chart. By induction on the size of the tree we see that a chart can only represent legal syntax trees.[6]

Example 2.28.

The tree from the previous example,

$$s_p(np_d(d_m,\, n_c),\ vp_t(v_e,\, np_p(n_f)))$$

can be represented by the following chart,

$$\begin{array}{rl}
[\, \mathsf{S} \to s_p[\mathsf{NP},\, \mathsf{VP}]\,;\ \phi_s\,;\ \phi_{np1},\, \phi_{vp}\,] & [\, \mathsf{D} \to d_m[]\,;\ \phi_d\,;\ ] \\
[\, \mathsf{NP} \to np_d[\mathsf{D},\, \mathsf{N}]\,;\ \phi_{np1}\,;\ \phi_d,\, \phi_{n1}\,] & [\, \mathsf{N} \to n_c[]\,;\ \phi_{n1}\,;\ ] \\
[\, \mathsf{VP} \to vp_t[\mathsf{V},\, \mathsf{NP}]\,;\ \phi_{vp}\,;\ \phi_v,\, \phi_{np2}\,] & [\, \mathsf{V} \to v_e[]\,;\ \phi_v\,;\ ] \\
[\, \mathsf{NP} \to np_p[\mathsf{N}]\,;\ \phi_{np2}\,;\ \phi_{n2}\,] & [\, \mathsf{N} \to n_f[]\,;\ \phi_{n2}\,;\ ]
\end{array}$$

where ϕ_s, ϕ_{np1}, ϕ_{vp}, ϕ_{np2}, ϕ_d, ϕ_{n1}, ϕ_v, ϕ_{n2} are matching linearizations.

▲

[6]In other words, a chart is sound by default.

Definition 2.29 (complete chart). A chart $\mathcal{C}$ is *complete* with respect to A and ϕ iff

- $[\,A\,;\,\phi\,] \subseteq \mathcal{C}$;
- $\mathcal{C}$ is complete with respect to B_i and ψ_i for each $[A \to f[\vec{B}]\,;\phi\,;\vec{\psi}] \in [A\,;\phi]$ and $1 \le i \le \delta_f$.

Lemma 2.30. *A complete (with respect to A and ϕ) chart $\mathcal{C}$ represents a tree t iff $t : A \Rightarrow \phi$.*

PROOF. In either direction, the tree t must be a legal syntax tree, and therefore we can say that it is of the form $t = f(\vec{t})$. We now use induction on the size of t.

($\Rightarrow$) If $\mathcal{C}$ represents t, then $[\,A \to f[\vec{B}]\,;\,\phi\,;\,\vec{\psi}\,] \in \mathcal{C}$ such that $[\![t]\!] = \phi$ and the subtrees $\vec{t}$ are represented by $\mathcal{C}$. The induction hypothesis says that $\vec{t} : \vec{B} \Rightarrow \vec{\psi}$, but since $t = f(\vec{t})$ and $\phi = f^{\circ}(\vec{\psi})$ we have that $t : A \Rightarrow \phi$.

($\Leftarrow$) If $t : A \Rightarrow \phi$, then $\phi = [\![t]\!] = [\![f(\vec{t})]\!] = f^{\circ}([\![\vec{t}]\!]) = f^{\circ}(\vec{\psi})$ for some $\vec{\psi}$ such that $\vec{t} : \vec{B} \Rightarrow \vec{\psi}$. But then t is represented by the item $[\,A \to f[\vec{B}]\,;\,\phi\,;\,\vec{\psi}\,]$ which is contained in $[\,A\,;\,\phi\,]$, and in $\mathcal{C}$ since the chart is complete. The induction hypothesis finally tells us that the subtrees $\vec{t}$ are represented by $\mathcal{C}$, and then the tree is also represented by $\mathcal{C}$.

□

Corollary 2.31. *The set $\mathcal{T}(S,\,w)$ of all syntactical analyses for an input string w can be represented by a complete chart with respect to S and w, where S is the starting category.*

In other words, a correct parsing algorithm for GCFG does not have to return anything more than a complete chart. Naturally, it is also useful to know that the algorithm always returns a finite chart. We do not dwell on this interesting subject in this thesis; we only note that the algorithms presented in chapter 4 all return finite charts.

If the chart is finite, extracting a tree that is represented by an item can be done in time linear in the size of the tree, with the assumption that items in the chart can be looked up in constant time.

Example 2.32.

The chart in example 2.28 is complete with respect to S and the input string *'many lions eat fish'*, since the represented tree is the only possible tree.

▲

2.6.3 Equivalence and simulation of grammars

There are (at least) two kinds of equivalence one can imagine, when talking about grammars. Weak equivalence is the most common, saying that two grammars G_1, G_2 are equivalent if they generate the same language, $\mathcal{L}(G_1) = \mathcal{L}(G_2)$. Unfortunately, for many purposes this notion of equivalence in not very useful. In our case we want to show that converting a grammar preserves something more than just the language; the conversion should also preserve the syntactical structures. Chomsky (1965) introduced the notion of *strong generative capacity* of a grammar G as the set of its syntactic structures, which in our setting can be the set $\mathcal{T}_G$ or $\mathcal{T}_G(S)$, depending on one's personal preferences. Unfortunately, this definition is not very useful and several others have been suggested; see e.g. Miller (1999) for a recent survey.

For our purposes the notion of *simulation* is well suited; of which the definition here is adapted from Chiang (2001).

Definition 2.33 (simulating interpretation). A *simulating interpretation* $\langle\!| \cdot |\!\rangle : \mathcal{T}_1 \to \mathcal{T}_2$ is a surjective mapping between two sets of trees, such that for each function symbol f occurring in $\mathcal{T}_1$,

$$\langle\!| f(\vec{t}) |\!\rangle \quad = \quad t[\vec{x}/\langle\!| \vec{t} |\!\rangle]$$

where $t[\vec{x}]$ is a tree in $\mathcal{T}_2$ whenever $\vec{x}$ are trees in $\mathcal{T}_2$.

In other words, a simulating interpretation transforms trees in a compositional way; it can always be defined by pattern matching on the function symbols (without taking cases of subtrees), as is done in the equation. Note that a simulating interpretation is efficiently computable in time proportional to the size of the input tree.

We say that $\langle\!| \cdot |\!\rangle$ is *trivial* if it is of one of the following forms,

- $\langle\!| f(x) |\!\rangle = x$, where f can take only one argument;
- $\langle\!| f(\vec{x}) |\!\rangle = g(\langle\!| \vec{y} |\!\rangle)$, where g is a function symbol in $\mathcal{T}_2$, and $\vec{y}$ is a permutation of $\vec{x}$.

In the first case, f is called a *coercion*. A trivial simulating interpretation can be specified by a bijective mapping (also written $\langle\!| \cdot |\!\rangle$) between $\mathcal{F}_1$ and $\mathcal{F}_2 \cup \{ - \}$,[7] where $\langle\!| f |\!\rangle = -$ when f is a coercion; together with a permutation $\theta : \mathcal{F}_1 \to \mathbb{N}^*$. We also use the notation $\langle\!| \cdot |\!\rangle$ for bijections between categories and linearizations.

[7]Recall that $\mathcal{F}$ is the set of function symbols in a grammar.

Example 2.34.

The following is a simulating interpretation between trees for the PMCFG grammar in figure 2.6, and trees for the original GF grammar in figure 2.4;

$$\begin{array}{rcl}
\langle\!|s_{p1}(x,\,y)|\!\rangle & = & s_p(\langle\!|x|\!\rangle,\,\langle\!|y|\!\rangle)\\
\langle\!|s_{p2}(x,\,y)|\!\rangle & = & s_p(\langle\!|x|\!\rangle,\,\langle\!|y|\!\rangle)\\
\langle\!|np_{d1}(x,\,y)|\!\rangle & = & np_d(\langle\!|x|\!\rangle,\,\langle\!|y|\!\rangle)\\
\langle\!|np_{d2}(x,\,y)|\!\rangle & = & np_d(\langle\!|x|\!\rangle,\,\langle\!|y|\!\rangle)\\
\langle\!|vp_{t1}(x,\,y)|\!\rangle & = & vp_t(\langle\!|x|\!\rangle,\,\langle\!|y|\!\rangle)\\
\langle\!|vp_{t2}(x,\,y)|\!\rangle & = & vp_t(\langle\!|x|\!\rangle,\,\langle\!|y|\!\rangle)\\
\langle\!|np_p(x)|\!\rangle & = & np_p(\langle\!|x|\!\rangle)\\
\langle\!|c|\!\rangle & = & c \qquad (c = d_a,\,d_m,\,n_c,\,n_f,\,v_e)
\end{array}$$

Furthermore, the interpretation is trivial and has no coercions.

▲

Definition 2.35 (simulation). A GCFG grammar G is a *(trivial) simulation* of another grammar G', if there is a (trivial) simulating interpretation, and a mapping $\langle\!|\cdot|\!\rangle$ between pairs of categories and linearizations such that if $t : A \Rightarrow \phi$ then $\langle\!|t|\!\rangle : A' \Rightarrow \phi'$, where $\langle\!|A\,;\,\phi|\!\rangle = A'\,;\,\phi'$. Furthermore, $\langle\!|S\,;\,w|\!\rangle = S'\,;\,w$ for all input strings w, where S and S' are the starting categories of G and G' respectively.

Note that if $A \to f[B]$ is a coercion in the simulation G, then $[\![f(t)]\!] = [\![t]\!]$ for all trees $t : B$, and $\langle\!|A\,;\,\phi|\!\rangle = \langle\!|B\,;\,\phi|\!\rangle$ whenever $A \Rightarrow \phi$. This also implies that any item for f is of the form $[\,A \to f[B]\,;\,\phi\,;\,\phi\,]$

Example 2.36.

Together with the simulating interpretation in the previous example, the following bijection between pairs of categories and linearizations constitutes a simulation by the PMCFG grammar of the original GF grammar;

$$\begin{array}{rcll}
\langle\!|\mathsf{S}\,;\,s=\alpha|\!\rangle & = & \mathsf{S}\,;\,\{\,s=\alpha\,\} & \\
\langle\!|X_1\,;\,s=\alpha|\!\rangle & = & X\,;\,\{\,s=\alpha\,;\,n=\mathsf{Sg}\,\} & (X=\mathsf{NP},\,\mathsf{D})\\
\langle\!|X_2\,;\,s=\alpha|\!\rangle & = & X\,;\,\{\,s=\alpha\,;\,n=\mathsf{Pl}\,\} & (X=\mathsf{NP},\,\mathsf{D})\\
\langle\!|Y\,;\,s_1=\alpha,\,s_2=\beta|\!\rangle & = & Y\,;\,\{\,s=[\,\mathsf{Sg}\Rightarrow\alpha\,;\,\mathsf{Pl}\Rightarrow\beta\,]\,\} & (Y=\mathsf{VP},\,\mathsf{N},\,\mathsf{V})
\end{array}$$

▲

Implications to parsing

Since the simulating interpretation is efficiently computable, a simulation can be used to parse a grammar in the obvious way; just use the simulating interpretation for translating back the parse trees.

If G trivially simulates G', there is a simple procedure transforming a complete chart for G, with respect to S and w, into a complete chart for G', with respect to S' and w.

Algorithm 2.37.

For each item $[\, A \to f[B]\, ;\, \phi\, ;\, \phi\,]$ such that f is a coercion, and for each item $\theta = [\, \ldots \to g[\ldots, A, \ldots]\, ;\, \ldots\, ;\, \ldots\,]$, add a copy of θ where A is replaced by B.

Then transform each item $[\, A \to f[\vec{B}]\, ;\, \phi\, ;\, \vec{\psi}\,]$ such that,

$$\begin{aligned} \langle\!|f|\!\rangle &= f' \neq - \\ \langle\!|A\, ;\, \phi|\!\rangle &= A'\, ;\, \phi' \\ \langle\!|\vec{B}\, ;\, \vec{\psi}|\!\rangle &= \vec{B}'\, ;\, \vec{\psi}' \end{aligned}$$

into the new item $[\, A' \to f'[\vec{B}']\, ;\, \phi'\, ;\, \vec{\psi}'\,]$.

▲

Lemma 2.38. *Transforming a complete chart $\mathcal{C}$, with respect to S and w, results in a complete chart $\mathcal{C}'$, with respect to S' and w.*

Proof. We have to show that $[\, A'\, ;\, \phi'\,]$ is contained in $\mathcal{C}'$ whenever $[\, A\, ;\, \phi\,]$ is contained in $\mathcal{C}$.

Since the simulating interpretation is surjective, each tree t' such that $t' : A' \Rightarrow \phi'$ will be the image $\langle\!|t|\!\rangle$ of some tree t such that $t : A \Rightarrow \phi$. But since $[\, A\, ;\, \phi\,]$ is contained in $\mathcal{C}$, each tree $t : A \Rightarrow \phi$ is represented by $\mathcal{C}$; which implies that each tree $t' : A' \Rightarrow \phi'$ will be represented by $\mathcal{C}'$, which means that $[\, A'\, ;\, \phi'\,]$ is contained in $\mathcal{C}$.

Finally, $\mathcal{C}$ is complete with respect to S' and w, since $\mathcal{C}$ is complete with respect to S and w, and $\langle\!|S\, ;\, w|\!\rangle = S'\, ;\, w$, $\mathcal{C}'$.

□

This procedure can be used to transform a chart returned by a parsing algorithm for the simulation, into a chart for the original grammar.

2.7 Summary

In this chapter we defined the basic notions for use in the rest of the thesis. Most importantly the grammar formalisms GF (with the important subclass *context-free* GF), GCFG and PMCFG were defined. Most of the material has been introduced by previous authors; it is only some things that are previously unseen. We stated two minor results which follow directly from the definitions; context-free GF is an instance of GCFG, and PMCFG is an instance of context-free GF.

In section 2.6 we discussed the representation of syntactical terms. We extended the notion of a shared forest for compactly representing a set of syntactical analyses, to the GCFG formalism. We also discussed when a grammar formalism, for which there are known parsing algorithms, can be used to parse grammars in another formalism. This was done by adapting the notion of grammar simulation from Chiang (2001).

Chapter 3

Reducing context-free GF to PMCFG

This chapter shows that context-free GF is strongly equivalent to PMCFG. This equivalence is shown by giving an algorithm converting context-free GF grammars into PMCFG grammars recognizing the same language; and by showing that parse results can be converted back efficiently.

The conversion algorithm consists of enumerating all parameter instantiations in a linearization, and then moving the instantiated parameters to the abstract categories. Enumerating all instantiations may lead to an exponential increase of the grammar size. Therefore two alternative conversion algorithms are given, which do not enumerate all possible instantiations, but instead try to only instantiate when it is necessary.

The main result of this chapter is the following theorem, which is a direct consequence of the conversions in sections 3.2 and 3.3.

Theorem 3.1. *Any context-free* GF *grammar can be transformed to an equivalent* PMCFG *grammar, which furthermore is a trivial simulation of the original grammar.*

Example 3.2.

Our example grammar throughout this chapter will be the canonical GF grammar in figure 2.4 on page 53.

▲

3.1 Paths and η-normal form

Definition 3.3 (path). A *path* is a sequence of record projections and table selections. The empty path is written ϵ, and $\sigma.r$ and $\sigma\,!\,\phi$ are paths if σ is a path.

A path that does not contain any argument variables x_i is called instantiated; in which case the selections ϕ can only be parameters. A non-instantiated path is called nested; this is because if a path contains an argument variable x_i, then that variable is always followed by a (possibly empty) path.

Note that we in the following equate nested tables and records with sets of path-value pairs. I.e. the nested linearization term

$$\begin{aligned} \{\, s &= [\,\mathsf{Sg} \Rightarrow \phi_1\,;\, \mathsf{Pl} \Rightarrow \phi_2\,]\,; \\ p &= \{\, n = \mathsf{Sg}\,;\, g = \mathsf{Utr}\,\}\,\} \end{aligned}$$

can also be written as a set of path-value pairs, or a flattened record,

$$\begin{aligned} \{\, s\,!\,\mathsf{Sg} &= \phi_1\,; \\ s\,!\,\mathsf{Pl} &= \phi_2\,; \\ p.n &= \mathsf{Sg}\,; \\ p.g &= \mathsf{Utr}\,\} \end{aligned}$$

Definition 3.4 (string path, parameter path). A linearization type T as well as a linearization ϕ can be partitioned into their *string paths* and *parameter paths*:

$$\begin{aligned} [T]^{\mathsf{Str}} &= \{\, \sigma : \mathsf{Str} \mid T.\sigma = \mathsf{Str}\,\} \\ [T]^{\mathsf{Par}} &= \{\, \sigma : P \mid T.\sigma = P \in \mathsf{Par}\,\} \\ [\phi]^{\mathsf{Str}} &= \{\, \sigma = \phi.\sigma \mid \phi.\sigma : \mathsf{Str}\,\} \\ [\phi]^{\mathsf{Par}} &= \{\, \sigma = \phi.\sigma \mid \phi.\sigma : P \in \mathsf{Par}\,\} \end{aligned}$$

Note that there are only a finite number of instantiated parameter records $\pi : [T]^{\mathsf{Par}}$, since there are only finitely many parameters.

Example 3.5.

For the terms $d_m : \mathsf{D}$ and $n_c : \mathsf{N}$ in the example grammar in figure 2.4 we have the following.

$$\begin{aligned}
[d_m^\circ]^{\mathsf{Str}} : [\mathsf{D}^\circ]^{\mathsf{Str}} &= \{\, s = \text{‘}\textit{many}\text{’} \,\} : \{\, s : \mathsf{Str} \,\} \\
[d_m^\circ]^{\mathsf{Par}} : [\mathsf{D}^\circ]^{\mathsf{Par}} &= \{\, n = \mathsf{Pl} \,\} : \{\, n : \mathsf{Num} \,\} \\
[n_c^\circ]^{\mathsf{Str}} : [\mathsf{N}^\circ]^{\mathsf{Str}} &= \{\, s\,!\,\mathsf{Sg} = \text{‘}\textit{lion}\text{’} \,;\ s\,!\,\mathsf{Pl} = \text{‘}\textit{lions}\text{’} \,\} \\
&: \{\, s\,!\,\mathsf{Sg} : \mathsf{Str} \,;\ s\,!\,\mathsf{Pl} : \mathsf{Str} \,\} \\
[n_c^\circ]^{\mathsf{Par}} : [\mathsf{N}^\circ]^{\mathsf{Par}} &= \{\,\} : \{\,\}
\end{aligned}$$

▲

Definition 3.6 (η-normal form). A linearization term ϕ of type T is in *η-normal form* if the structure follows the structure of its linearization type:

- If T is a record type, $\{\, r_1 : T_1 \,;\ \ldots\,;\ r_n : T_n \,\}$, then ϕ is a record $\{\, r_1 = \phi_1 \,;\ \ldots\,;\ r_n = \phi_n \,\}$ where each subterm ϕ_i is in η-normal form.
- If T is a table type $P \Rightarrow T_0$ and $P = \{\, p_1, \ldots, p_n \,\}$, then ϕ is a table $[\, p_1 \Rightarrow \phi_1 \,;\ \ldots\,;\ p_n \Rightarrow \phi_n \,]$ where each subterm ϕ_i is in η-normal form.
- If T is a basic linearization type, Str or $P \in \mathsf{Par}$, then ϕ is called a *leaf*.

3.2 Converting to table normal form

Definition 3.7 (table normal form). A GF linearization is in *table normal form* if it is of the form

$$\begin{aligned}
f &: B_1 \times \cdots \times B_\delta \to A \\
f^\circ(x_1, \ldots, x_\delta) &= [\, \pi_1 \Rightarrow \phi_1 \,;\ \ldots\,;\ \pi_n \Rightarrow \phi_n \,]\,!\,\xi
\end{aligned}$$

and the following hold:

- ξ contains all parameter paths of the arguments $x_1, \ldots, x_\delta$;

$$\xi = \left\langle [x_1]^{\mathsf{Par}}, \ldots, [x_\delta]^{\mathsf{Par}} \right\rangle$$

- Each π_k is a possible parameter instantiation of $x_1, \ldots, x_\delta$;

$$\pi_k : [B_1^\circ]^{\mathsf{Par}} \times \cdots \times [B_\delta^\circ]^{\mathsf{Par}}$$

- $\pi_1, \ldots, \pi_n$ is an exhaustive enumeration of instantiations;

$$\{\, \pi_1, \ldots, \pi_n \,\} = [B_1^\circ]^{\mathsf{Par}} \times \cdots \times [B_\delta^\circ]^{\mathsf{Par}}$$

- Each ϕ_k is in η-normal form where the leaves are either parameters or concatenations of constant strings and instantiated string paths.

The following algorithm converts any GF linearization in canonical form[1] into table normal form.

Algorithm 3.8.

Given a GF function with a context-free backbone;

$$\begin{aligned} f &: B_1 \times \cdots \times B_\delta \to A \\ f^\circ(x_1, \ldots, x_\delta) &= \phi \end{aligned}$$

convert the canonical linearization ϕ to table normal form by applying the following two steps;

- First, add the outer table as in the definition of table normal form;

$$\begin{aligned} f^\circ(x_1, \ldots, x_\delta) &= [\, \pi_1 \Rightarrow \phi \,; \ldots ; \pi_n \Rightarrow \phi \,] \,!\, \xi \\ \xi &= \langle [x_1]^{\mathsf{Par}}, \ldots, [x_\delta]^{\mathsf{Par}} \rangle \\ \pi_k &: [B_1^\circ]^{\mathsf{Par}} \times \cdots \times [B_\delta^\circ]^{\mathsf{Par}} \end{aligned}$$

- Second, for each instantiation π_k, convert ϕ to ϕ_k, by repeating the following substitution until there are no parameter paths left;
  - Substitute each term $x_i.\sigma$, where σ is an instantiated parameter path, by its π_k-instantiation $(\pi_k)_i.\sigma$.

▲

Note that the normal form of a linearization, can very well lead to an exponential increase of the size of the linearization. The reason is that the outer table $[\, \pi_1 \Rightarrow \phi_1 \,; \ldots ; \pi_n \Rightarrow \phi_n \,]$ has a number of rows proportional to the total number of parameters occurring in the GF linearization.

Lemma 3.9. *Algorithm 3.8, together with the standard computation rules, yields an equivalent linearization in table normal form.*

PROOF. Assume that the resulting linearization is not in table normal form. Then there must be some ϕ_k which is not in the form described in definition 3.7. Now ϕ_k can not contain any instantiated parameter paths, since they are substituted by the algorithm.

The only possibility for ϕ_k is therefore to contain nested paths. Then there must be a "least" nested path σ, occurring as $x_i.\sigma$, where σ does not contain any nested paths itself. But then σ can neither contain nested paths, nor instantiated parameter paths, meaning that σ is not nested. We have a contradiction. □

[1]The canonical form is defined in section 2.3.7.

Example 3.10.

There are three linearizations in the example that are not in table normal form, and this is how they look after conversion;

$$
\begin{array}{rcl}
s_p^\circ(x,\,y) & = & [\,\mathsf{Sg} \Rightarrow \{\, s = x.s \,\cdot\, y.s\,!\,\mathsf{Sg}\,\}\,;\\
 & & \;\mathsf{Pl} \;\Rightarrow \{\, s = x.s \,\cdot\, y.s\,!\,\mathsf{Pl}\,\}\,]\;!\;x.n\\
np_d^\circ(x,\,y) & = & [\,\mathsf{Sg} \Rightarrow \{\, s = x.s \,\cdot\, y.s\,!\,\mathsf{Sg}\,;\; n = \mathsf{Sg}\,\}\,;\\
 & & \;\mathsf{Pl} \;\Rightarrow \{\, s = x.s \,\cdot\, y.s\,!\,\mathsf{Pl}\,;\; n = \mathsf{Pl}\,\}\,]\;!\;x.n\\
vp_t^\circ(x,\,y) & = & [\,\mathsf{Sg} \Rightarrow \{\, s = [\,\mathsf{Sg} \Rightarrow x.s\,!\,\mathsf{Sg} \,\cdot\, y.s\,;\\
 & & \qquad\qquad\quad \mathsf{Pl} \;\Rightarrow x.s\,!\,\mathsf{Pl} \,\cdot\, y.s\,]\,\}\,;\\
 & & \;\mathsf{Pl} \;\Rightarrow \{\, s = [\,\mathsf{Sg} \Rightarrow x.s\,!\,\mathsf{Sg} \,\cdot\, y.s\,;\\
 & & \qquad\qquad\quad \mathsf{Pl} \;\Rightarrow x.s\,!\,\mathsf{Pl} \,\cdot\, y.s\,]\,\}\,]\;!\;y.n
\end{array}
$$

▲

3.3 Converting to a PMCFG grammar

To get a PMCFG grammar, we have to get rid of the parameters in some way; and this we do by moving them to the abstract syntax. Each table row $\pi_k \Rightarrow \phi_k$ resulting from algorithm 3.8 will then give rise to a unique function symbol $\hat{f}$ with linearization ϕ_k.

Algorithm 3.11.

Given the context-free backbone of a GF grammar where all linearizations are in table normal form, create a grammar with the following categories, function symbols and linearizations:

- For each category A and each instantiated parameter record $\pi : [A^\circ]^{\mathsf{Par}}$, create a new category $\hat{A} = A[\pi]$. The linearization type is the same as the string paths of the original linearization type, $\hat{A}^\circ = [A^\circ]^{\mathsf{Str}}$.
- For each syntax rule $f : B_1 \times \cdots \times B_\delta \to A$, and all new categories $\hat{A}$, $\hat{B}_1, \ldots, \hat{B}_\delta$, create a new syntax rule $\hat{f} : \hat{B}_1 \times \cdots \times \hat{B}_\delta \to \hat{A}$; where $\hat{f}$ is a unique function symbol, $\hat{f} = f[\hat{B}_1 \cdots \hat{B}_\delta \to \hat{A}]$.
- For each linearization function,

$$f^\circ(x_1,\,\ldots,\,x_\delta) \;=\; [\,\pi_1 \Rightarrow \phi_1\,;\,\ldots\,;\,\pi_n \Rightarrow \phi_n\,]\,!\,\xi$$

and each table row $\pi_k \Rightarrow \phi_k$, create a new linearization function for $\hat{f}$;

$$
\begin{array}{rcl}
\hat{f}^\circ(x_1,\ldots,x_\delta) & = & [\phi_k]^{\mathsf{Str}}\\
\hat{A} & = & A[[\phi_k]^{\mathsf{Par}}]\\
\hat{A}_i & = & A_i[(\pi_k)_i]
\end{array}
$$

where we by $(\pi_k)_i$ mean the ith component of π_k.

The resulting grammar is a PMCFG grammar, since all linearizations are records of strings.

▲

A trivial simulation

Recalling the definition of simulation in section 2.6.3, we define a trivial simulating interpretation, $\langle\!\langle f(\vec{t}) \rangle\!\rangle = g(\langle\!\langle \vec{t} \rangle\!\rangle)$, where g is the function symbol such that $f = \hat{g}$. A mapping between pairs of categories and linearizations can be defined as $\langle\!\langle A\,;\, \phi \rangle\!\rangle = B\,;\, \psi$ where B is the category such that $A = \hat{B} = B[\pi]$, and $\psi = \phi \cup \pi$. With these two functions we can state the following lemma.

Lemma 3.12. *The resulting* PMCFG *grammar is a trivial simulation of the original* GF *grammar.*

PROOF. We have to show that $t : A \Rightarrow \phi$ implies that $\langle\!\langle t \rangle\!\rangle : B \Rightarrow \psi$ where $\langle\!\langle A\,;\, \phi \rangle\!\rangle = B\,;\, \psi$. We proceed by induction on the size of the tree $t = f(\vec{t})$, where $f : \vec{A} \rightarrow A$.

Assume that $t : A \Rightarrow \phi$. But from the algorithm we know that there are g, B and π such that $g : \vec{B} \rightarrow B$, $f = \hat{g}$, $A = \hat{B} = B[\pi]$ and $\psi = \phi \cup \pi$. Now, $f(\vec{t}) : A \Rightarrow \phi$ implies that $\vec{t} : \vec{A} \Rightarrow \vec{\phi}$, which by the induction hypothesis is equivalent to $\langle\!\langle \vec{t} \rangle\!\rangle : \vec{B} \Rightarrow \vec{\psi}$, which in turn implies that $\langle\!\langle t \rangle\!\rangle : B \Rightarrow \psi$.

□

Example 3.13.

Figure 3.1 shows how the example grammar looks like after conversion to PMCFG. Note that the grammar is equivalent to example 2.20, modulo renaming of categories, functions and labels.

▲

3.4 Non-deterministic reduction

Another possible conversion is to use a non-deterministic substitution algorithm. This can also in some cases reduce the size of the resulting PMCFG grammar, when argument parameters are not mentioned in the original linearizations.

Algorithm 3.14.

Assume the following abstract syntax rule, together with its linearization function:

$$\begin{array}{rcl} f & : & B_1 \times \cdots \times B_\delta \rightarrow A \\ f^\circ(x_1, \ldots, x_\delta) & = & \phi \end{array}$$

Repeat the following non-deterministic substitution until there are no instantiated parameter paths left, accumulating the parameter records $\pi_1, \ldots, \pi_\delta$:

Categories and linearization types

$$\begin{array}{rcll}
\hat{X}_1 & = & X[n = \mathsf{Sg}] & (X = \mathsf{D},\, \mathsf{NP}) \\
\hat{X}_2 & = & X[n = \mathsf{Pl}] & (X = \mathsf{D},\, \mathsf{NP}) \\
\hat{X} & = & X[] & (X = \mathsf{S},\, \mathsf{VP},\, \mathsf{V},\, \mathsf{N}) \\
\widehat{\mathsf{S}}^\circ,\, \widehat{\mathsf{D}}_1^\circ,\, \widehat{\mathsf{D}}_2^\circ,\, \widehat{\mathsf{NP}}_1^\circ,\, \widehat{\mathsf{NP}}_2^\circ & = & \{\, s : \mathsf{Str} \,\} & \\
\widehat{\mathsf{N}}^\circ,\, \widehat{\mathsf{V}}^\circ,\, \widehat{\mathsf{VP}}^\circ & = & \{\, s\,!\,\mathsf{Sg} : \mathsf{Str}\,;\; s\,!\,\mathsf{Pl} : \mathsf{Str} \,\} &
\end{array}$$

Functions

$$\begin{array}{rcll}
\hat{s}_{pi} & : & \widehat{\mathsf{NP}}_i \times \widehat{\mathsf{VP}} \to \widehat{\mathsf{S}} & (i = 1,\, 2) \\
\widehat{np}_{di} & : & \widehat{\mathsf{D}}_i \times \widehat{\mathsf{N}} \to \widehat{\mathsf{NP}}_i & (i = 1,\, 2) \\
\widehat{np}_p & : & \widehat{\mathsf{N}} \to \widehat{\mathsf{NP}}_2 & \\
\widehat{vp}_{ti} & : & \widehat{\mathsf{V}} \times \widehat{\mathsf{NP}}_i \to \widehat{\mathsf{VP}} & (i = 1,\, 2) \\
\hat{d}_a & : & \widehat{\mathsf{D}}_1 & \\
\hat{d}_m & : & \widehat{\mathsf{D}}_2 & \\
\hat{n}_f,\, \hat{n}_c & : & \widehat{\mathsf{N}} & \\
\hat{v}_e & : & \widehat{\mathsf{V}} &
\end{array}$$

Linearization functions

$$\begin{array}{rcl}
\hat{s}_{p1}^\circ(x,\, y) & = & \{\, s = x.s \cdot y.s\,!\,\mathsf{Sg} \,\} \\
\hat{s}_{p2}^\circ(x,\, y) & = & \{\, s = x.s \cdot y.s\,!\,\mathsf{Pl} \,\} \\
\widehat{np}_{d1}^\circ(x,\, y) & = & \{\, s = x.s \cdot y.s\,!\,\mathsf{Sg} \,\} \\
\widehat{np}_{d2}^\circ(x,\, y) & = & \{\, s = x.s \cdot y.s\,!\,\mathsf{Pl} \,\} \\
\widehat{np}_p^\circ(x) & = & \{\, s = x.s\,!\,\mathsf{Pl} \,\} \\
\widehat{vp}_{t1}^\circ(x,\, y) & = & \{\, s\,!\,\mathsf{Sg} = x.s\,!\,\mathsf{Sg} \cdot y.s\,; \\
 & & \quad s\,!\,\mathsf{Pl} = x.s\,!\,\mathsf{Pl} \cdot y.s \,\} \\
\widehat{vp}_{t2}^\circ(x,\, y) & = & \text{the same as } \widehat{vp}_{t1}^\circ(x,\, y) \\
\hat{d}_a^\circ & = & \{\, s = \text{‘}\textit{a}\text{’} \,\} \\
\hat{d}_m^\circ & = & \{\, s = \text{‘}\textit{many}\text{’} \,\} \\
\hat{n}_c^\circ & = & \{\, s\,!\,\mathsf{Sg} = \text{‘}\textit{lion}\text{’}\,;\; s\,!\,\mathsf{Pl} = \text{‘}\textit{lions}\text{’} \,\} \\
\hat{n}_f^\circ & = & \{\, s\,!\,\mathsf{Sg} = \text{‘}\textit{fish}\text{’}\,;\; s\,!\,\mathsf{Pl} = \text{‘}\textit{fish}\text{’} \,\} \\
\hat{v}_e^\circ & = & \{\, s\,!\,\mathsf{Sg} = \text{‘}\textit{eats}\text{’}\,;\; s\,!\,\mathsf{Pl} = \text{‘}\textit{eat}\text{’} \,\}
\end{array}$$

Figure 3.1: Example grammar after conversion to PMCFG.

- Substitute each instantiated parameter path $x_i.\sigma : P$ with any $p \in P$, such that the unification $\pi_i \sqcup \{ \sigma = p \}$ is defined.[2] Update π_i with the result of the unification.

Supposing that the final substituted linearization is ψ, we can add the following rule for the new function symbol $\hat{f}$:

$$\begin{aligned} \hat{f} &: \hat{B}_1 \times \cdots \times \hat{B}_\delta \rightarrow \hat{A} \\ \hat{f}^\circ(x_1, \ldots, x_\delta) &= [\psi]^{\mathsf{Str}} \\ \hat{A} &= A[[\psi]^{\mathsf{Par}}] \\ \hat{B}_i &= B_i[\pi_i] \end{aligned}$$

▲

The algorithm is non-deterministic, and we get the final grammar by finding all solutions for each function symbol f. This terminates since there are only a finite number of parameters.

3.4.1 Coercions between categories

There is a difference between algorithm 3.14 and the previous algorithms 3.8 + 3.11; if an argument parameter $x_i.\sigma$ is not mentioned in ϕ (i.e. if the linearization is erasing), then there will be no σ-row in the constraint record π_i. This means that the new category $\hat{B}_i = B_i[\pi_i]$ will only contain a subrecord of $B_i[[\psi_i]^{\mathsf{Par}}]$, where ϕ_i is a linearization of type B_i°. This problem can be solved by introducing coercion functions between $B_i[\pi_i]$ and $B_i[[\psi_i]^{\mathsf{Par}}]$.

Algorithm 3.15.

Consider two syntax rules resulting from algorithm 3.14,

$$\begin{aligned} \hat{f} &: \cdots \times \hat{B}_1 \times \cdots \rightarrow \hat{A} \\ \hat{g} &: \cdots \rightarrow \hat{B}_2 \end{aligned}$$

where $\hat{B}_1 = B[\pi_1]$ and $\hat{B}_2 = B[\pi_2]$. If π_1 is a subrecord of π_2, add the coercion function $\hat{c} = c[\pi_1\pi_2]$:

$$\begin{aligned} \hat{c} &: \hat{B}_2 \rightarrow \hat{B}_1 \\ \hat{c}^\circ(x) &= x \end{aligned}$$

▲

Applying algorithm 3.14 and then algorithm 3.15 results in a grammar that is a trivial simulation of the original grammar. This is not difficult to see, since the coercion functions will be coercions in the simulating interpretation.

[2]Recall that we defined a simplistic variant of record unification in section 2.1.2; $\pi_1 \sqcup \pi_2 = \pi_1 \cup \pi_2$ whenever there is no r such that $\pi_1.r \neq \pi_2.r$.

Example 3.16.

One function symbol gets a linearization from algorithm 3.15 that differs from figure 3.1; the functions $\widehat{vp}_{t1}$ and $\widehat{vp}_{t2}$ get merged into one function $\widehat{vp}_t$,

$$\begin{aligned} \widehat{vp}_t &: \widehat{\mathsf{V}} \times \widehat{\mathsf{NP}} \to \widehat{\mathsf{V}} \\ \widehat{vp}_t^\circ(x, y) &= \{\, s\,!\,\mathsf{Sg} = x.s\,!\,\mathsf{Sg} \cdot y.s\,; \\ &\qquad s\,!\,\mathsf{Pl} \;= x.s\,!\,\mathsf{Pl} \cdot y.s \,\} \end{aligned}$$

where $\widehat{\mathsf{NP}} = \mathsf{NP}[]$. This yields coercions for the more specific types $\widehat{\mathsf{NP}}_1 = \mathsf{NP}[n = \mathsf{Sg}]$ and $\widehat{\mathsf{NP}}_2 = \mathsf{NP}[n = \mathsf{Pl}]$.

$$\begin{aligned} \hat{c}_i &: \widehat{\mathsf{NP}}_i \to \widehat{\mathsf{NP}} \qquad (i = 1, 2) \\ \hat{c}_i^\circ(x) &= x \end{aligned}$$

▲

3.5 Tables with anonymous variables

In full GF it is possible to have anonymous tables of the form $[\,\_ \Rightarrow \phi\,]$, meaning that the value of the parameter is uninteresting. In canonical form such a table will have the form $[\, p_1 \Rightarrow \phi\,; \ldots ; p_n \Rightarrow \phi \,]$. The algorithms presented so far will then result in n copies of ϕ in each resulting linearization. Here we show how to reduce this overhead, in a way similar to the REGULUS compiler (Rayner et al., 2001), which compiles limited unification-based grammars into context-free grammars.

We assume that anonymous tables are written as $[\,\_ \Rightarrow \phi\,]$. This can be accomplished either by transforming each table $[\, p_1 \Rightarrow \phi_1\,; \ldots ; p_n \Rightarrow \phi_n \,]$ such that $\phi_1 = \cdots = \phi_n$; or by changing the canonical form compiler into leaving anonymous tables alone.[3]

3.5.1 Constraints and anonymous variables

The non-deterministic substitution in algorithm 3.14 remains almost the same. But now we have the possibility of reducing a selection from an anonymous table $[\,\_ \Rightarrow \phi\,]\,!\,(x_i.\sigma)$ directly to ϕ, without updating the constraint record π_i at all. This means that there are two conflicting behaviors if $x_i.\sigma$ is an instantiated parameter path; either substitute it by any $p \in P$ and update π_i, or reduce to ϕ without updating π_i. In either case, the final result will be ϕ, but in the former we get several solutions, one for each $p \in P$. Therefore, we should try to reduce $[\,\_ \Rightarrow \phi\,]\,!\,x_i.\sigma$ directly whenever possible. The best way to do this is to reduce a term from the inside; i.e. when considering a term $\phi\,!\,(x_i.\sigma)$, first reduce ϕ and

[3]This is already implemented as an option in the current implementation of GF.

check whether it is an anonymous table. If it is, reduce without updating π_i, otherwise substitute by some $p \in P$ and update π_i.

Now, assume the following initial rule

$$\begin{aligned} f &: B_1 \times \cdots \times B_\delta \to A \\ f^\circ(x_1, \ldots, x_\delta) &= \phi \end{aligned}$$

After constraint reduction of ϕ we will get ψ, together with the constraints $\pi_1, \ldots, \pi_\delta$. From this we can deduce the following rule for the new function symbol $\hat{f}$:

$$\begin{aligned} \hat{f} &: \hat{B}_1 \times \cdots \times \hat{B}_\delta \to \hat{A} \\ \hat{f}^\circ(x_1, \ldots, x_\delta) &= [\psi]^{\mathsf{Str}} \\ \hat{A} &= A[[\psi]^{\mathsf{Par}} ; \Sigma] \\ \hat{B}_i &= B_i[\pi_i ; \emptyset] \\ \Sigma &= \{ \sigma \mid \psi.\sigma = [\,\_ \Rightarrow \psi'\,] \} \end{aligned}$$

Note that the new categories $\hat{A}$ and $\hat{B}_i$ consists of the parameter paths, together with a set Σ. This set contains the paths for all anonymous tables of the resulting term.

3.5.2 More coercion functions

With anonymous tables, algorithm 3.15 for creating coercions has to be extended.

Algorithm 3.17.

Given two rules,

$$\begin{aligned} \hat{f} &: \cdots \times \hat{B}_1 \times \cdots \to \hat{A} \\ \hat{g} &: \ldots \to \hat{B}_2 \end{aligned}$$

where $\hat{B}_1 = B[\pi_1 ; \emptyset]$, $\hat{B}_2 = B[\pi_2 ; \Sigma]$ and $\hat{B}_1 \neq \hat{B}_2$. If π_1 is a subrecord of π_2, then we can add the coercion $\hat{c} = c[\pi_1 \pi_2 \Sigma]$,

$$\begin{aligned} \hat{c} &: \hat{B}_2 \to \hat{B}_1 \\ \hat{c}^\circ(x) &= \{ \sigma_1 = x.\sigma_1^* ; \ldots ; \sigma_n = x.\sigma_n^* \} \end{aligned}$$

where σ_i^* is created from σ_i by the following substitution; whenever $\sigma_i = \sigma\,!\,p.\sigma'$ and $\sigma \in \Sigma$, i.e. a prefix of σ_i is in Σ, replace p by $\_$ in σ_i^*.

▲

Note that there can be more than one substitution; if $\sigma \in \Sigma$ and $\sigma\,!\,\_.\sigma' \in \Sigma$, then $\sigma_i = \sigma\,!\,p.\sigma'\,!\,p'$ will be replaced by $\sigma_i^* = \sigma\,!\,\_.\sigma'\,!\,\_$.

Example 3.18.

The original (non-canonical) rule for n_f contains an anonymous table,

$$\begin{aligned} n_f &: \mathsf{N} \\ n_f^\circ &= \{\, s = [\,\_ \Rightarrow \text{‘}\textit{fish}\text{’}\,]\,\} \end{aligned}$$

The non-deterministic reduction then results in the following rule,

$$\begin{aligned} \hat{n}_f &: \widehat{\mathsf{N}}_s \\ \hat{n}_f^\circ &= \{\, s\,!\,\_ = \text{‘}\textit{fish}\text{’}\,\} \end{aligned}$$

where $\widehat{\mathsf{N}}_s = \mathsf{N}[\,;\,s]$, since s is the path for the only anonymous table. The rest of the grammar results in the same PMCFG grammar as figure 3.1 augmented with example 3.16. Finally we get a coercion from the “anonymous noun” $\widehat{\mathsf{N}}_s = \mathsf{N}[\,;\,s]$ to the standard noun $\widehat{\mathsf{N}} = \mathsf{N}[]$,

$$\begin{aligned} \hat{c} &: \mathsf{N}[\,;\,s] \to \mathsf{N}[] \\ \hat{c}^\circ(x) &= \{\, s\,!\,\mathsf{Sg} = x.s\,!\,\_\,;\ s\,!\,\mathsf{Pl} = x.s\,!\,\_\,\} \end{aligned}$$

▲

3.6 Summary

The main result of this chapter is that any context-free GF grammar can be transformed to an equivalent PMCFG grammar. Furthermore, the resulting grammar is a simulation, meaning that it can be used for the purpose of parsing the original context-free GF grammar. The translation works by first instantiating all tables and table selections, and converting them to records and record projections; and then all parameters are moved to the abstract syntax. This means among other things that a category in the original grammar can be split into a number of distinct categories in the resulting grammar; and that a function in the original grammar can be split, or duplicated, into several functions with different typings.

Since a simulating PMCFG grammar always exists, context-free GF can be seen as a nice front-end for PMCFG, in the same way as GENERALIZED PHRASE-STRUCTURE GRAMMAR (Gazdar et al., 1985) can be seen as a nice front-end for CFG.

As noted in section 3.2, the translation from GF to PMCFG can lead to an exponential increase of the grammar size. Therefore two alternative translation algorithms were given that can in some cases reduce the increase of grammar size. The main idea of these variants is that it is not always necessary to instantiate every possible table and parameter, and in these cases a number of similar grammar rules (and categories) can be merged into one single rule (and category), together with simple coercion functions between the merged categories and the original categories.

Chapter 4

Parsing algorithms for context-free GF and PMCFG

This chapter investigates a number of tabular parsing algorithms for context-free GF and PMCFG, all with polynomial time complexity. Starting with a general passive algorithm similar to the one given by Seki et al. (1991), several different modifications are suggested.

The search space can be reduced by approximating the PMCFG grammar by an over-generating CFG. Afterwards the context-free parse results can be translated back into PMCFG parse results, which have to be checked for correctness since the CFG is over-generating.

Another alternative is to use an active algorithm, in the spirit of the context-free Earley (1970) algorithm. We give two active algorithms; one recognizing the linearization rows of a rule in a fixed order, and another recognizing rows incrementally according to the order in which they occur in the input. Both top-down and bottom-up prediction strategies are investigated.

All suggested algorithms, except for the last incremental version, require that the PMCFG grammar is nonerasing; therefore we give an algorithm for removing erasingness from a grammar.

A note on erasing grammars

The algorithms in sections 4.2, 4.3 and 4.4 only work for *nonerasing* grammars. In section 4.5 it is discussed how to handle grammars where linearization arguments are deleted. The final algorithm in section 4.6 works for erasing and suppressing grammars directly.

A note on items and charts

The parse items defined in the algorithms in this chapter are strictly not items in the sense of definition 2.25 in section 2.6.2. But it is not difficult to convert the parse items resulting from an algorithm to items satisfying the definition.

The soundness and completeness results of the algorithms can then be used to show that the transformed chart is complete according to definition 2.29.

A running example

Example 4.1.

Throughout this chapter we will use the following example grammar when exemplifying the algorithms.

$$\begin{array}{rcl} \mathsf{S} \to f[\mathsf{A}] & := & s = \mathsf{A}.p\ \mathsf{A}.q \\ \mathsf{A} \to g[\mathsf{A}^1, \mathsf{A}^2] & := & p = \mathsf{A}^1.p\ \mathsf{A}^2.p, \\ & & q = \mathsf{A}^1.q\ \mathsf{A}^2.q \\ \mathsf{A} \to ac[] & := & p = `a\text{'},\ \ q = `c\text{'} \\ \mathsf{A} \to bd[] & := & p = `b\text{'},\ \ q = `d\text{'} \end{array}$$

This grammar generates the language,

$$\mathcal{L} \ = \ \{\, s\,\theta(s) \mid s \in (a \cup b)^* \,\}$$

where θ is a homomorphic mapping satisfying $\theta(a) = c$ and $\theta(b) = d$.

This language is a kind of "copy-morphism" language, since the second occurrence of s is transformed through the homomorphism θ. Some strings that are accepted by the grammar are '*ac*', '*bd*', '*abcd*', '*badc*', '*aacc*', '*bbdd*', and '*abbacddc*'.

▲

4.1 Ranges

The idea of ranges is taken from RANGE CONCATENATION GRAMMAR (RCG; Boullier, 2000a,b). But instead of using pairs of input positions as in the RCG formalism, we use sets of pairs. The reason for this is that a GF/PMCFG grammar can have reduplication of strings.

Definition 4.2 (range). Given an input string w, the *universal range* $\mathcal{R}_w$ is the set of all pairs of input positions, $\mathcal{R}_w = \{\, (i, j) \mid 0 \leq i \leq j \leq |w| \,\}$. A *range* ρ is a nonempty subset of $\mathcal{R}_w$. Concatenation is a partial operation on ranges,

$$\rho_1 \cdot \rho_2 \quad = \quad \{\, (i, k) \mid (i, j) \in \rho_1,\ (j, k) \in \rho_2 \,\}$$

whenever the resulting set is non-empty.

There is a partial function from a string to a range,

$$\langle s \rangle^w \quad = \quad \{\, (i, j) \mid s = w_{i+1} \dots w_j \,\}$$

whenever s is a substring of w. If the input string w is known, we simply write $\langle s \rangle$. We write $i \dots j$ for the range $\langle w_{i+1} \dots w_j \rangle$. A string s is an *image* of a range ρ if $\rho = \langle s \rangle$. In the sequel we only consider *string-equivalent* ranges, i.e. ranges that have an image. The string-equivalent ranges are closed under concatenation, and form a partition of $\mathcal{R}_w$. There are only $\mathcal{O}(|w|^2)$ string-equivalent ranges (instead of $2^{|w|}$ ranges in total) and they can be stored in constant space by only remembering the first pair. Concatenation of string-equivalent ranges can be done in constant time by creating a "multiplication table" of size $\mathcal{O}(|w|^4)$ before-hand.

For string-equivalent ranges ρ we write w^ρ for the string $w_{i+1} \dots w_j$ whenever $(i,\, j) \in \rho$. This means that $w^{i \dots j} = w_{i+1} \dots w_j$. Note that $\langle w^\rho \rangle = \rho$; and $w^{\langle s \rangle} = s$ whenever s is a substring of w. The *empty range* $\langle \epsilon \rangle$ matches the empty string and is equal to $\{\, (i,\, i) \mid 0 \leq i \leq |w| \,\}$. If Γ is a data structure (such as a list, a record or a tree) containing one or more ranges, we write w^Γ for the data structure where all occurrences of a range ρ in Γ are replaced by the string w^ρ; and if ϕ is a data structure containing strings, we write $\langle \phi \rangle$ for the data structure where all strings are replaced by matching ranges.

If Γ or ϕ are *incomplete* in the sense that they contain as yet unbound variables, then these variables are left unchanged in w^Γ and $\langle \phi \rangle$. If Γ or ϕ are in the context of a rule $A \to f[\vec{B}] := \psi$, then argument variables B_i are considered as unbound variables.

Example 4.3.

Given the input string $w =$ '*abaabaa*', the following are examples of ranges;

$$\begin{aligned}
\rho_1 &= \langle abaa \rangle &&= \{\, (0,\, 4),\, (3,\, 7) \,\} && (= 0 \dots 4 = 3 \dots 7) \\
\rho_2 &= \langle aaba \rangle &&= \{\, (2,\, 6) \,\} && (= 2 \dots 6) \\
\rho_3 &= \langle ab \rangle &&= \{\, (0,\, 2),\, (3,\, 5) \,\} \\
\rho_4 &= \langle a \rangle &&= \{\, (0,\, 1),\, (2,\, 3),\, (3,\, 4),\, (5,\, 6),\, (6,\, 7) \,\}
\end{aligned}$$

Now, given that,

$$\Gamma \quad = \quad \{\, s = \rho_3 \cdot \rho_4 \cdot \rho_4\, ;\ t = \rho_3 \cdot x \cdot \rho_3 \,\}$$

where x is a variable, we have that,

$$w^\Gamma \quad = \quad \{\, s = \text{'}abaa\text{'}\, ;\ t = \text{'}ab\text{'} \cdot x \cdot \text{'}ab\text{'} \,\}$$

▲

4.1.1 Range-restriction

We say that a linearization is *string-concatenative* if the only linearization operation involving strings is concatenation; GF and PMCFG are both string-concatenative formalisms. The notion of *range-restriction* is only meaningful for string-concatenative linearizations.

Definition 4.4 (range-restriction). A GF linearization ϕ can be *range-restricted* by an input string w. In the resulting linearization $\langle \phi \rangle$, each string constant s is replaced by the range $\langle s \rangle$, and string concatenation is replaced by range concatenation.

Note that range-restriction is a partial operation since $\langle s \rangle$ is a partial function.

A linearization can contain at most $\mathcal{O}(|\phi|)$ constant strings, and since each of these takes $\mathcal{O}(|w|)$ time to restrict, the following lemma is trivial.

Lemma 4.5. *Range-restricting a* GF *linearization ϕ by w can be done in time $\mathcal{O}(|\phi| \cdot |w|)$.*

This lemma is noted only to make sure that range-restriction is not the main part of the time complexity for a parsing algorithm.

Example 4.6.

The following rule in the example grammar,

$$A \rightarrow ac[] \quad := \quad p = \text{‘}a\text{’},\ q = \text{‘}c\text{’}$$

can be range-restricted by the input string ‘*abbacddc*’, resulting in the following rule,

$$A \rightarrow ac[] \quad := \quad p = \{\ (0,1), (3,4)\ \}\ ,\ \ q = \{\ (4,5), (7,8)\ \}$$

▲

4.1.2 Ranges and linear GF grammars

Recall from section 2.4.4 that a grammar is linear if no part of any argument variable occurs more than once in a linearization. In a linear GF grammar each record projection for a linearization variable $B_i.r$ occurs at most once.

A non-linear grammar has a rule where some argument projection occurs twice, i.e. $A \rightarrow f[\vec{B}] := \phi_1\ B_i.r\ \phi_2\ B_i.r\ \phi_3$. In the final linearized string w, the first occurrence of $B_i.r$ represents some substring $w_i \ldots w_j$ and the second occurrence represents another substring $w_{i'} \ldots w_{j'}$. This is the reason why ranges are represented as *sets* of index pairs, and not just index pairs.

A linear grammar does not have this reduplication of arguments, which means that it is not necessary to represent ranges as sets. Instead we can use a representation as pairs of indices (i, j), where $0 \leq i \leq j \leq |w|$, as done in RCG. Concatenation is defined as $(i, j) \cdot (j', k) = (i, k)$ whenever $j = j'$. We call this representation *simple ranges*, as opposed to the previous set-representation. When using simple ranges, there are only some small things to note;

- The partial function $\langle s \rangle$ from strings to ranges, becomes a many-valued function;
- This implies that range-restriction becomes a non-deterministic operation.

All algorithms from this chapter work on simple ranges with only slight modifications. The only difference is that each occurrence of $\rho = \langle s \rangle$ should be replaced by $\rho \in \langle s \rangle$. As an example, the inference rule 4.12 of section 4.4 becomes (after simplification),

$$\frac{[\, R\,;\, \Gamma,\, r = (i,\, j) \bullet s\; \alpha,\, \phi\,;\, \vec{\Gamma}\,]}{[\, R\,;\, \Gamma,\, r = (i,\, k) \bullet \alpha,\, \phi\,;\, \vec{\Gamma}\,]} \;\left\{\; s = w_{j+1} \ldots w_k \right.$$

Example 4.7.

The example grammar is linear, meaning that we can use simple ranges instead. Range-restricting the example grammar with the same input string, now results in the following four rules,

$$\begin{array}{lcl}
A \to ac[] & := & p = (0,\, 1)\,,\;\; q = (4,\, 5) \\
A \to ac[] & := & p = (3,\, 4)\,,\;\; q = (4,\, 5) \\
A \to ac[] & := & p = (0,\, 1)\,,\;\; q = (7,\, 8) \\
A \to ac[] & := & p = (3,\, 4)\,,\;\; q = (7,\, 8)
\end{array}$$

▲

4.2 Polynomial parsing for context-free GF

Given the range definitions above, it is straightforward to describe a simple bottom-up parsing algorithm for context-free GF grammars. This algorithm is a natural extension of the CKY algorithm and similar to the one described by Seki et al. (1991), only this one is more general since it also works for a more general formalism than PMCFG.

The parse items for a rule $A \to f[\vec{B}] := \phi$ are of the form $[\, A \to f[\vec{B}]\,;\, \Gamma\,;\, \vec{\Gamma}\,]$, where Γ and Γ_i are range-restricted linearizations. The interpretation is that there is some tree $t = f(\vec{t}) : A$ such that $[\![t]\!] = w^{\Gamma}$ and $[\![\vec{t}]\!] = w^{\vec{\Gamma}}$.

Combine

$$\frac{[\,B_1\,;\,\Gamma_1\,] \quad \ldots \quad [\,B_\delta\,;\,\Gamma_\delta\,]}{[\,A \to f[\vec{B}]\,;\,\Gamma\,;\,\vec{\Gamma}\,]} \left\{ \begin{array}{l} A \to f[\vec{B}] := \phi \\ \Gamma = \langle\phi\rangle\,[\vec{B}/\vec{\Gamma}] \end{array} \right. \tag{4.1}$$

First we range-restrict the linearization ϕ, and then substitute each argument variable B_i by its range-restricted linearization Γ_i.

Since the grammar is nonerasing, all Γ_i are contained in Γ. From this we can define a ranking of parse items where each antecedent is less than the consequent. This implies that the inference rule can be implemented by the generalized cky deduction engine (algorithm 2.4). A simple item ranking can be defined as $r([\,A\,;\,\Gamma\,]) = |w^{\Gamma}|$, which works as long as there are no coercions in the grammar. Recall from section 2.6.3 that a coercion is a rule of the form $A \to f[B]$ with $f^{\circ}(x) = x$.

When we want to prove completeness, the ranking r above works well for grammars without coercions. In the general case we can define a ranking based on the size of the corresponding minimal tree instead;[1]

$$d([\,A \to f[\vec{B}]\,;\,\Gamma\,;\,\vec{\Gamma}\,]) \quad = \quad \min\{\, |t| \mid t = f(\vec{t}) : A,\ [\![t]\!] = w^{\Gamma},\ [\![\vec{t}]\!] = w^{\vec{\Gamma}} \,\}$$

Theorem 4.8. *Inference rule 4.1 is sound and complete.*

Proof. Soundness follows from the fact that the antecedents $[\,B_i : \Gamma_i\,]$ say that there are trees $t_i : B_i$ such that $[\![t_i]\!] = \Gamma_i$. Then the tree $t = f(t_1, \ldots, t_\delta)$ is type-correct and has linearization $[\![t]\!] = \phi[\vec{B}/w^{\vec{\Gamma}}]$; and since $\Gamma = \langle\phi\rangle\,[\vec{B}/\vec{\Gamma}]$, the consequent item has a correct interpretation.

Completeness follows from the fact that the only way to infer an item is by the Combine rule, and then the size $|t| = |f(t_1, \ldots, t_\delta)| > |t_i|$ for all trees t, including the minimal tree.

□

Example 4.9.

Given the example grammar and an input string $w =$ '*acbd*', the final goal item is $[\,S\,;\,\Gamma_w\,]$, where $\Gamma_w = \{\, s = \langle w\rangle \,\}$ and $\langle w\rangle = 0\ldots4 = \{\,(0,\,4)\,\}$. Here is an example derivation using inference rule 4.1,

1	$[\,A \to ac[]\,;\,\Gamma_{a,c}\,;\ \,]$	Combine
1′	$[\,A\,;\,\Gamma_{a,c}\,]$	
2	$[\,A \to bd[]\,;\,\Gamma_{b,d}\,;\ \,]$	Combine
2′	$[\,A\,;\,\Gamma_{b,d}\,]$	
3	$[\,A \to g[A^1,\,A^2]\,;\,\Gamma_{ab,cd}\,;\,\Gamma_{a,c},\,\Gamma_{b,d}\,]$	Combine (1'), (2')
3′	$[\,A\,;\,\Gamma_{ab,cd}\,]$	
4	$[\,S \to f[A]\,;\,\Gamma_w\,;\,\Gamma_{ab,cd}\,]$	Combine (3')
4′	$[\,S\,;\,\Gamma_w\,]$	

[1]Recall from section 2.6.1 that the size $|t|$ of a tree $t = f(\vec{t})$ is equal to $1 + |\vec{t}|$.

where the range records Γ_i are as follows,

$$\begin{aligned}
\Gamma_{a,c} &= \{\, p = \langle a \rangle \;;\; q = \langle c \rangle \,\} \\
\Gamma_{b,d} &= \{\, p = \langle b \rangle \;;\; q = \langle d \rangle \,\} \\
\Gamma_{ab,cd} &= \{\, p = \langle ab \rangle \;;\; q = \langle cd \rangle \,\} \\
\Gamma_{w} &= \{\, s = \langle abcd \rangle \,\}
\end{aligned}$$

▲

Theorem 4.10. *For a context-free* GF *grammar, the final chart is finite and polynomial in the length of the input. Thus the algorithm terminates in polynomial time in the length of the input.*

Proof. To get an upper bound of the number of items we observe that any linearization type $[\![A]\!]$ in the grammar G can only contain a finite number d_A of occurrences of Str. In an item of a given rule $A \to f[\vec{B}] := \phi$, there are $d_A + \sum d_{B_i}$ different ranges. For each range there are $\mathcal{O}(n^2)$ possibilities, where $n = |w|$ is the length of the input. Thus, there are $\mathcal{O}(n^{2(d_A + \sum d_{B_i})})$ possible items for the given rule. The space complexity is $\mathcal{O}(|\mathcal{R}|n^{2e})$, where $e = \max_{A \to f[\vec{B}]}(d_A + \sum d_{B_i})$.

For the time complexity we note that since all information available in the antecedent items and the side conditions also exist in the consequent, each item will only be inferred once. This in turn means that the time for inferring one item is constant, given that the calculation in the side condition is constant. Thus, the time complexity is equal to the space complexity.

□

For PMCFG grammars, the upper bound for space and time complexity can be tightened to $\mathcal{O}(|\mathcal{R}|n^{e+1})$ by inspecting the structure of the linearization records. Informally, the reason is that the daughter strings in an item are not independent of each other; e.g. in a linearization $r = A.r\ B.s\ C.t$, the leftmost position of $B.s$ must be equal to the rightmost position of $A.r$, and the rightmost position of $B.s$ must be equal to the leftmost position of $C.t$. For a more detailed explanation, see Seki et al. (1991).

4.2.1 An active version of the algorithm

The algorithm above suffers from the same problem as similar context-free parsing algorithms do; for the inference rule to apply, we must find δ matching items, which can take long time for large δ. The standard solution is to introduce partial results, giving us the possibility to match one item at the time. The items now look like $[\, A \to f[\vec{B} \bullet \vec{B}'] \,;\, \Gamma \,;\, \vec{\Gamma} \,]$ with $|\vec{B}| = |\vec{\Gamma}|$, where the categories $\vec{B}$ to the left of the dot are found with linearizations $\vec{\Gamma}$. The linearization Γ is a partially instantiated linearization. When $\vec{B}'$ is empty, the item is *passive* and Γ is fully instantiated. In this case we can write $[\, B \,;\, \Gamma \,]$ for the passive item $[\, B \to g[\ldots \bullet] \,;\, \Gamma \,;\, \ldots \,]$.

Predict

$$\frac{}{[\, A \to f[\bullet \vec{B}] \,;\, \Gamma \,;\,]} \quad \begin{cases} A \to f[\vec{B}] := \phi \\ \Gamma = \langle \phi \rangle \end{cases} \tag{4.2}$$

Prediction converts each grammar rule to a range-restricted equivalent active item.

Combine

$$\frac{[\, A \to f[\vec{B} \bullet B_k,\, \vec{B}'] \,;\, \Gamma \,;\, \vec{\Gamma} \,] \quad [\, B_k \,;\, \Gamma_k \,]}{[\, A \to f[\vec{B},\, B_k \bullet \vec{B}'] \,;\, \Gamma' \,;\, \vec{\Gamma},\, \Gamma_k \,]} \quad \big\{ \; \Gamma' = \Gamma[B_k/\Gamma_k] \tag{4.3}$$

Here we substitute only one argument variable B_k by its range-restricted linearization Γ_k.

Example 4.11.

We use the same grammar and input string $w =$ '*abcd*' as in example 4.9; and get the following derivation,

1	$[\, S \to f[\bullet A] \,;\, \Gamma^0_w \,;\,]$	Predict
2	$[\, A \to g[\bullet A^1,\, A^2] \,;\, \Gamma^0_{ab,cd} \,;\,]$	Predict
3	$[\, A \to ac[\bullet] \,;\, \Gamma_{a,c} \,;\,]$	Predict
4	$[\, A \to bd[\bullet] \,;\, \Gamma_{b,d} \,;\,]$	Predict
5	$[\, A \to g[A^1 \bullet A^2] \,;\, \Gamma^1_{ab,cd} \,;\, \Gamma_{a,c} \,]$	Combine (2), (3)
6	$[\, A \to g[A^1,\, A^2 \bullet] \,;\, \Gamma_{ab,cd} \,;\, \Gamma_{a,c},\, \Gamma_{b,d} \,]$	Combine (4), (6)
7	$[\, S \to f[A \bullet] \,;\, \Gamma_w \,;\,]$	Combine (1), (7)

where the range records Γ_w, $\Gamma_{a,c}$, $\Gamma_{b,d}$, $\Gamma_{ab,cd}$ are as in example 4.9; Γ^0_w, $\Gamma^0_{ab,cd}$ are uninstantiated as in their corresponding grammar rules; and $\Gamma^1_{ab,cd}$ is partially instantiated,

$$\Gamma^1_{ab,cd} \quad = \quad \{\, p = \langle a \rangle \;\; A^2.p \,;\, q = \langle c \rangle \;\; A^2.q \,\}$$

▲

Soundness and completeness

We here give arguments of why the active algorithm is correct. To show correctness of the inference rules 4.2 and 4.3, we first have to give an interpretation of parse items. The interpretation of an item $[\, A \to f[\vec{B} \bullet \vec{B}'] \,;\, \Gamma \,;\, \vec{\Gamma} \,]$, is that there is an *open* tree $t = f(\vec{t}, \vec{t}') : A$, for which $[\![t]\!] = w^{\Gamma}$ and $[\![\vec{t}]\!] = w^{\vec{\Gamma}}$. Recall from section 2.6.1 that an open tree may contain argument variables B_i somewhere. In this case we can be more specific; each tree in $\vec{t}$ is instantiated, and each tree t_i in $\vec{t}'$ is equal to the argument variable B_i. Note that for passive items the tree t is instantiated, and the interpretation coincides with the interpretation of the passive algorithm.

For the completeness proofs we assume that the associated grammar rule is;

$$A \to f[\vec{B}] \quad := \quad \phi$$

Soundness of Predict follows from the fact that,

$$[\![f(\vec{B})]\!] = \phi = w^{\langle \phi \rangle} = w^{\Gamma}$$

Soundness of Combine follows from $[\![f(\vec{t},\, B_k,\, \vec{B})]\!] = w^{\Gamma}$ and that there is a tree $t_k : B_k$ such that $[\![t_k]\!] = w^{\Gamma_k}$. Then $[\![f(\vec{t},\, t_k,\, \vec{B})]\!] = w^{\Gamma}[B_k / w^{\Gamma_k}] = w^{\Gamma'}$ by the side condition of Combine.

Completeness can be shown similarly to the completeness proof for inference rule 4.1; the ranking is based on the size of the minimal incomplete tree matching the interpretation, where the size of an uninstantiated subtree is zero. Now, given an item $[\, A \to f[\vec{B} \bullet \vec{B}'] \,;\, \Gamma \,;\, \vec{\Gamma}\,]$, there are two possibilities;

- Either $\vec{B}$ is empty, in which case the item is inferred by prediction;
- Otherwise $\vec{B}$ ends with B_k and the item is inferred by combining. The trees $t(\vec{t},\, B_k,\, \vec{B}') : A$ and $t_k : B_k$ for the antecedents are both smaller than the consequent tree $t(\vec{t},\, t_k,\, \vec{B}') : A$, since $|B_k| = 0$ and $|t_k| > 0$.

4.3 Parsing through context-free approximation

In this section we show how to parse a PMCFG grammar by converting it to a context-free grammar, and then recovering the PMCFG chart from the context-free chart. The recovery consists of two steps: first the context-free chart is converted to an equivalent PMCFG chart; then the items in that chart are combined for discontinuous constituents.

Decorated context-free grammars

The theory in this section gets much simpler if we use a variant of context-free grammars, where the rules are decorated with extra information.

Definition 4.12 (decorated rule). A *decorated* context-free rule is of the form $f : A \to \beta$, where f is the name of the rule, and each non-terminal B in β can have some associated information i, written as a superscript of the non-terminal in question, B^i.

Example 4.13.

Some rules from the example English grammar in section 1.3.5 might look like this as decorated context-free rules;

$$\begin{array}{rcccll}
s_p & : & \mathsf{S} & \rightarrow & \mathsf{NP}^1 & \mathsf{VP}^2 \\
np_d & : & \mathsf{NP} & \rightarrow & \mathsf{D}^1 & \mathsf{N}^2 \\
np_p & : & \mathsf{NP} & \rightarrow & \mathsf{N}^1 & \\
vp_t & : & \mathsf{VP} & \rightarrow & \mathsf{V}^1 & \mathsf{NP}^2
\end{array}$$

▲

Any parsing algorithm for CFG can be trivially transformed to a parsing algorithm for decorated grammars, simply by ignoring the decorations when looking up matching rules and parse items. From now on we assume that the parsing algorithm returns a chart of items $[^\circ\, i - j\,;\ f : A \rightarrow \beta\,]$, as described in section 2.2.2. The only difference is that the name of the rule is added to the item, and that the categories in β might be decorated.

4.3.1 Creating a context-free approximation

The first step is to convert the PMCFG grammar to a decorated context-free grammar. This is done by splitting the linearization record of each PMCFG rule into several context-free rules.

Algorithm 4.14.

From the PMCFG rule,

$$A \rightarrow f[\vec{B}] \quad := \quad r_1 = \alpha_1, \ldots, r_n = \alpha_n$$

create n decorated context-free rules $f : A.r_k \rightarrow \alpha_k$, for $1 \leq k \leq n$.

▲

Note that we do not have to change α_k at all, since the variable-free notation for PMCFG is already decorated.

The final context-free grammar will be over-generating, meaning that all sentences recognized by the original PMCFG grammar will also be recognized by the decorated CFG. The reason for this is that the resulting CFG cannot constrain several occurrences of an argument category to represent the same item; an example of this is shown in examples 4.15–4.18. That the CFG is over-generating means in turn that a sound and complete context-free parsing algorithm will still be complete, but unsound.

Example 4.15.

The example grammar looks like follows, when converted to a decorated CFG,

$$\begin{array}{rclll}
f & : & S.s & \to & A.p \quad A.q \\
g & : & A.p & \to & A.p^1 \quad A.p^2 \\
g & : & A.q & \to & A.q^1 \quad A.q^2 \\
ac & : & A.p & \to & \text{`}a\text{'} \\
ac & : & A.q & \to & \text{`}c\text{'} \\
bd & : & A.p & \to & \text{`}b\text{'} \\
bd & : & A.q & \to & \text{`}d\text{'}
\end{array}$$

▲

4.3.2 Converting context-free items to PMCFG items

Creating PMCFG pre-items

After parsing we get a chart of context-free parse items which are converted to PMCFG *pre-items* as follows.

Algorithm 4.16.

Each decorated context-free item,

$$[^\circ\, j - k\,;\; f : A.r \to \beta\,]$$

matching the rule $A \to f[\vec{B}]$, is converted to a PMCFG *pre-item*,[2]

$$[^\bullet\, A \to f[\vec{B}]\,;\; r = j \ldots k\,;\; \vec{\Gamma}\,]$$

where $\vec{\Gamma}$ is a partition of the daughters in β such that,

$$\Gamma_i \quad = \quad \{\, r' = \rho \mid B_i.r' \in \beta,\, B_i.r' \Rightarrow^* w^\rho \,\}$$

where $B_i.r' \Rightarrow^* w^\rho$ is defined by the following equivalence;

$$X \Rightarrow^* w^{j \ldots k} \quad \textbf{iff} \quad [^\circ\, j - k\,;\; g : X \to \gamma\,]$$

▲

Example 4.17.

After parsing the input string $w =$ '*abcd*' using the decorated grammar, we get the following context-free chart;

$$\begin{array}{lll}
1^\circ & [^\circ\, 0 - 1\,;\; ac : A.p \to \text{`}a\text{'}\,] & \\
2^\circ & [^\circ\, 1 - 2\,;\; bd : A.p \to \text{`}b\text{'}\,] & \\
3^\circ & [^\circ\, 2 - 3\,;\; ac : A.q \to \text{`}c\text{'}\,] & \\
4^\circ & [^\circ\, 3 - 4\,;\; bd : A.q \to \text{`}d\text{'}\,] & \\
5^\circ & [^\circ\, 0 - 2\,;\; g : A.p \to A.p^1\; A.p^2\,] & \text{from } (1^\circ) \text{ and } (2^\circ) \\
6^\circ & [^\circ\, 2 - 4\,;\; g : A.q \to A.q^1\; A.q^2\,] & \text{from } (3^\circ) \text{ and } (4^\circ) \\
7^\circ & [^\circ\, 1 - 3\,;\; f : S.s \to A.p\; A.q\,] & \text{from } (2^\circ) \text{ and } (3^\circ) \\
8^\circ & [^\circ\, 0 - 4\,;\; f : S.s \to A.p\; A.q\,] & \text{from } (5^\circ) \text{ and } (6^\circ)
\end{array}$$

[2]Recall that $j \ldots k$ is the range $\langle w_{j+1} \ldots w_k \rangle$.

These context-free items are then converted to PMCFG pre-items,

$1^\bullet$	$[^\bullet\, A \to ac[\,]\,;\, \Gamma_a\,;\ ]$
$2^\bullet$	$[^\bullet\, A \to bd[\,]\,;\, \Gamma_b\,;\ ]$
$3^\bullet$	$[^\bullet\, A \to ac[\,]\,;\, \Gamma_c\,;\ ]$
$4^\bullet$	$[^\bullet\, A \to bd[\,]\,;\, \Gamma_d\,;\ ]$
$5^\bullet$	$[^\bullet\, A \to g[A^1,\, A^2]\,;\, \Gamma_{ab}\,;\, \Gamma_a,\, \Gamma_b\,]$
$6^\bullet$	$[^\bullet\, A \to g[A^1,\, A^2]\,;\, \Gamma_{cd}\,;\, \Gamma_c,\, \Gamma_d\,]$
$7^\bullet$	$[^\bullet\, S \to s[A]\,;\, \Gamma_{b,c}\,;\, \Gamma_b,\, \Gamma_c\,]$
$8^\bullet$	$[^\bullet\, S \to s[A]\,;\, \Gamma_{ab,cd}\,;\, \Gamma_{ab},\, \Gamma_{cd}\,]$

where the range records are as follows;

$$\begin{array}{rcll} \Gamma_x & = & \{\, p = \langle x\rangle \,\} & (x = a,\ b,\ ab) \\ \Gamma_y & = & \{\, q = \langle y\rangle \,\} & (y = c,\ d,\ cd) \\ \Gamma_{x,y} & = & \{\, p = \langle x\rangle\ ;\ q = \langle y\rangle \,\} & \end{array}$$

▲

Combining pre-items

Several pre-items can finally be combined to full items with the following single inference rule.

Combine

$$\frac{[^\bullet\, R\,;\, r_1 = \rho_1\,;\, \vec{\Gamma}_1\,] \quad \ldots \quad [^\bullet\, R\,;\, r_n = \rho_n\,;\, \vec{\Gamma}_n\,]}{[\, R\,;\, r_1 = \rho_1,\, \ldots,\, r_n = \rho_n\,;\, \vec{\Gamma}\,]} \left\{\ \vec{\Gamma} = \vec{\Gamma}_1 \sqcup \ldots \sqcup \vec{\Gamma}_n \right. \tag{4.4}$$

Each consequent daughter Γ_i is equal to the unification of the antecedents' corresponding daughters $\Gamma_{1,i} \sqcup \ldots \sqcup \Gamma_{n,i}$, where we use the simplistic unification defined in section 2.1.2.

Unfortunately, this algorithm is unsound since the underlying parsing algorithm gives unsound items. This means that the chart might contain incorrect items.

Example 4.18.

Applying this inference rule to the pre-items from the example grammar and input string $w =$ '*abcd*', results in the following chart;

1	$[\, A \to ac[\,]\,;\, \Gamma_{a,c}\,;\ ]$	Combine ($1^\bullet$), ($3^\bullet$)
2	$[\, A \to bd[\,]\,;\, \Gamma_{b,d}\,;\ ]$	Combine ($2^\bullet$), ($4^\bullet$)
3	$[\, A \to g[A^1,\, A^2]\,;\, \Gamma_{ab,cd}\,;\, \Gamma_{a,c},\, \Gamma_{b,d}\,]$	Combine ($5^\bullet$), ($6^\bullet$)
4	$[\, S \to s[A]\,;\, s = \langle bc\rangle\ ;\, \Gamma_{b,c}\,]$	Combine ($7^\bullet$)
5	$[\, S \to s[A]\,;\, \Gamma_w\,;\, \Gamma_{ab,cd}\,]$	Combine ($8^\bullet$)

where $\Gamma_w = \{\, s = \langle w\rangle \,\}$ and Γ_x is like in the previous example. Now note that item (4) is not correct, since the grammar does not recognize the string '*bc*'.

▲

Marking for correctness

The algorithm is complete though, since the underlying algorithm is complete, meaning that the chart contains all correct items. So, what we can do is mark the correct items in the chart until there are no more correct items to mark.

Algorithm 4.19.

Repeat the following until there are no more items to mark:

Mark an item $[\, A \to f[\vec{B}]\, ;\, \Gamma\, ;\, \vec{\Gamma}\,]$ as correct, if there are marked items $[\, B_i\, ;\, \Gamma_i\,]$ for each $1 \leq i \leq \delta_f$

▲

Alternatively, add the following inference rule to the one in the previous section, where we use $[\,\cdot\,]^\dagger$ to mark items.

Mark

$$\frac{[\, A \to f[\vec{B}]\, ;\, \Gamma\, ;\, \vec{\Gamma}\,] \qquad [\, B_1\, ;\, \Gamma_1\,]^\dagger \quad \ldots \quad [\, B_\delta\, ;\, \Gamma_\delta\,]^\dagger}{[\, A \to f[\vec{B}]\, ;\, \Gamma\, ;\, \vec{\Gamma}\,]^\dagger} \tag{4.5}$$

Recall that $[\, B\, ;\, \Gamma\,]$ means the passive item $[\, B \to \ldots\, ;\, \Gamma\, ;\, \ldots\,]$. Note that this inference rules can be implemented with the generalized CKY deduction engine (algorithm 2.4).

Example 4.20.

Now the incorrect item (4) in the example chart,

$$[\, S \to s[A]\, ;\, s = \langle bc\rangle\ ;\, \Gamma_{b,c}\,]$$

will never get marked, since there is no item $[\, A\, ;\, \Gamma_{b,c}\,]$ in the chart. The final chart consists of the items (1), (2), (3) and (5); note that this chart is equivalent to the chart that results in example 4.9.

▲

4.3.3 Soundness and completeness

Soundness and completeness of the algorithm follows from the fact that the algorithm in section 4.2 is sound and complete. Note that the inference rule to mark items is almost equivalent to inference rule 4.1 in section 4.2. The only difference is that here the value of Γ is precomputed in the unmarked item, but in section 4.2 we have to compute the value on every invocation of the rule.

So, the algorithm is sound as long as the invariant $w^\Gamma = \phi[\vec{B}/w^{\vec{\Gamma}}]$ is correct in the unmarked item. This follows from soundness of the context-free algorithm and that inference rule 4.4 maintains the invariant. And the algorithm is complete since the algorithm for combining pre-items is complete.

4.3.4 An active version of the algorithm

The two inference rules Combine and Mark can be divided into four active rules, if we introduce dotted items. The items are of the following forms;

$$[\, A \to f[\vec{B}]\,;\, \Gamma \bullet \phi\,;\, \vec{\Gamma}\,] \qquad\qquad [\, A \to f[\vec{B}]\,;\, \Gamma\,;\, \vec{\Gamma} \bullet \vec{\Gamma}'\,]$$

The interpretation of items of the first form is that the linearizations before the dot has been recognized, and the linearizations after the dot remains to be recognized. The interpretation of the second form of items is that the daughters before the dot are correct, and the daughters after the dot have to be checked for correctness.

Pre-Predict

$$\frac{}{[\, A \to f[\vec{B}]\,;\ \bullet \phi\,;\, \vec{\Gamma}_\emptyset\,]} \left\{\ A \to f[\vec{B}] := \phi \right. \tag{4.6}$$

where by $\vec{\Gamma}_\emptyset$ is meant an δ_f-element sequence of empty records.

Pre-Combine

$$\frac{[\, R\,;\, \Gamma \bullet r = \alpha,\, \phi\,;\, \vec{\Gamma}\,] \quad [^\bullet\, R\,;\ r = \rho\,;\, \vec{\Gamma}'\,]}{[\, R\,;\, \Gamma,\, r = \rho \bullet \phi\,;\, \vec{\Gamma}''\,]} \left\{\ \vec{\Gamma}'' = \vec{\Gamma} \sqcup \vec{\Gamma}' \right. \tag{4.7}$$

If we are looking for the row r, and there is a matching pre-item such that the daughters can be unified, we can move the dot forward. Note that the linearization α is not used, since it is already recognized by the pre-item.

Mark-Predict

$$\frac{[\, R\,;\, \Gamma \bullet\ ;\, \vec{\Gamma}\,]}{[\, R\,;\, \Gamma\,;\ \bullet \vec{\Gamma}\,]} \tag{4.8}$$

When we are finished with incorporating pre-items, we can start to mark for correctness.

Mark-Combine

$$\frac{[\, A \to f[\vec{B}]\,;\, \Gamma\,;\, \vec{\Gamma} \bullet \Gamma_i,\, \vec{\Gamma}'\,] \quad [\, B_i\,;\, \Gamma_i\,]}{[\, A \to f[\vec{B}]\,;\, \Gamma\,;\, \vec{\Gamma},\, \Gamma_i \bullet \vec{\Gamma}'\,]} \tag{4.9}$$

where we write $[\, B\,;\, \Gamma\,]$ for any passive item $[\, B \to \dots\,;\, \Gamma\,;\, \dots \bullet\,]$. If we want to mark daughter B_i for correctness, and there is a correct passive item for B_i, we can move the dot forward.

4.4 Active parsing of PMCFG

In this section we give an active algorithm for pmcfg grammars, which parses a pmcfg grammar directly without having to use any context-free approximation.

Parse items

The parse items of a rule,

$$A \to f[\vec{B}] \quad := \quad \psi,\, r = \beta\;\alpha,\, \phi$$

are of the form,

$$[\, A \to f[\vec{B}]\,;\, \Gamma,\, r = \rho \bullet \alpha,\, \phi\,;\, \vec{\Gamma}\,]$$

The informal meaning is that *i*) all rows ψ have been recognized as Γ; *ii*) the sequence β has been recognized as the range ρ; and that *iii*) for each argument B_i occurring in ψ or β, there is a passive item $[\, B_i\,;\, \Gamma_i\,]$.

For passive items $[\, R\,;\, \Gamma \bullet\ ;\, \vec{\Gamma}\,]$, this amounts to the same interpretation as in the passive algorithm in section 4.2.

Example 4.21.

The following are examples of parse items for the g-rule from the example grammar, and the input string '*abcd*',

1	$[\, A \to g[A^1, A^2]\,;\, p = \langle\epsilon\rangle \bullet A^1.p\ A^2.p,\ q = A^1.q\ A^2.q\,;\, \Gamma_\emptyset, \Gamma_\emptyset\,]$
2	$[\, A \to g[A^1, A^2]\,;\, p = \langle ab\rangle\,,\ q = \langle c\rangle \bullet A^2.q\,;\, \Gamma_{a,c}, \Gamma_{b,d}\,]$
3	$[\, A \to g[A^1, A^2]\,;\, p = \langle ab\rangle\,,\ q = \langle cd\rangle \bullet\ ;\, \Gamma_{a,c}, \Gamma_{b,d}\,]$

where $\Gamma_{a,c}$, $\Gamma_{b,d}$ are as in example 4.9.

The first item has not found anything at all; while the second item has found the full p row spanning the range $\langle ab\rangle = 0\ldots 2$, and is in the middle of recognizing the q row. The last item is a passive item which has found both rows spanning $\langle ab\rangle = 0\ldots 2$ and $\langle cd\rangle = 2\ldots 4$; this item can also be written $[\, A\,;\, \Gamma_{ab,cd}\,]$. ▲

Inference rules

There are four inference rules, and the following shorthands are used;

- By $[\, A\,;\, \Gamma\,]$ we mean any passive item $[\, A \to \ldots\,;\, \Gamma \bullet\ ;\, \ldots]$;
- By $\vec{\Gamma}_\emptyset$ we mean a δ_f-element sequence of empty records;
- By Γ_i we mean the ith element of the sequence $\vec{\Gamma}$, and by $\vec{\Gamma}[i := \Gamma']$ we mean that Γ_i is replaced by Γ'.

Predict

$$\frac{}{[\, A \to f[\vec{B}]\,;\, r = \langle\epsilon\rangle \bullet \alpha,\, \phi\,;\, \vec{\Gamma}_\emptyset\,]} \;\Big\{\; A \to f[\vec{B}] := r = \alpha,\, \phi \tag{4.10}$$

Prediction is very crude; it just converts each rule to a parse item saying that the empty range is found; which of course is always true.

Complete

$$\frac{[\,R\,;\,\Gamma,\,r=\rho\bullet\,,\,r'=\alpha,\,\phi\,;\,\vec{\Gamma}\,]}{[\,R\,;\,\Gamma,\,r=\rho,\,r'=\langle\epsilon\rangle\bullet\alpha,\,\phi\,;\,\vec{\Gamma}\,]} \tag{4.11}$$

Completion applies when a row in the linearization record has been found, and moves the dot into the next row. There it says that the empty range is found.

Scan

$$\frac{[\,R\,;\,\Gamma,\,r=\rho\bullet s\;\alpha,\,\phi\,;\,\vec{\Gamma}\,]}{[\,R\,;\,\Gamma,\,r=\rho'\bullet\alpha,\,\phi\,;\,\vec{\Gamma}\,]}\;\Big\{\;\rho'=\rho\cdot\langle s\rangle \tag{4.12}$$

Scanning applies when the next item to read is a string constant.

Combine

$$\frac{[\,R\,;\,\Gamma,\,r=\rho\bullet B_i.r'\;\alpha,\,\phi\,;\,\vec{\Gamma}\,]\quad[\,B_i\,;\,\Gamma'\,]}{[\,R\,;\,\Gamma,\,r=\rho'\bullet\alpha,\,\phi\,;\,\vec{\Gamma}[i:=\Gamma']\,]}\;\begin{cases}\rho'=\rho\cdot\Gamma'.r'\\ \Gamma_i\subseteq\Gamma'\end{cases} \tag{4.13}$$

This is the only complicated rule, applying when the next item is an r' label of argument B_i. It succeeds if there is a matching passive item $[\,B_i\,;\,\Gamma'\,]$, for which $\Gamma_i\subseteq\Gamma'$. By this is meant that the linearization Γ' of the passive item is consistent with what was previously known about argument B_i; and since Γ' comes from a passive item, it is instantiated and we do not have to use unification; a subset check suffices.

Comparing to traditional algorithms for context-free grammars, such as the ones in section 2.2.2, there is one extra rule Complete. This rule acts like a kind of prediction for subsequent rows in a linearization record.

Example 4.22.

A derivation for the example grammar and the input string '*abcd*' is shown in figure 4.1.

▲

4.4.1 Different prediction strategies

The Predict rule given above is very crude, adding every rule in the grammar as a hypothesis. This can be a serious problem for large grammars. Chart parsing algorithms for context-free grammars have better prediction strategies, as described in section 2.2.2. In this section we extend those strategies to PMCFG parsing.

1	$[\, R_{ac}\,;\ p = \langle\epsilon\rangle \bullet \text{‘}a\text{’},\ q = \text{‘}c\text{’}\,;\ ]$	Predict
2	$[\, R_{ac}\,;\ p = \langle a\rangle \bullet ,\ q = \text{‘}c\text{’}\,;\ ]$	Scan (1)
3	$[\, R_{ac}\,;\ p = \langle a\rangle ,\ q = \langle\epsilon\rangle \bullet \text{‘}c\text{’}\,;\ ]$	Complete (2)
4	$[\, R_{ac}\,;\ p = \langle a\rangle ,\ q = \langle c\rangle \bullet\ ;\ ]$	Scan (3)
$4'$	$[\, A\,;\ \Gamma_{a,c}\,]$	
5	$[\, R_{bd}\,;\ p = \langle\epsilon\rangle \bullet \text{‘}b\text{’},\ q = \text{‘}d\text{’}\,;\ ]$	Predict
6	$[\, R_{bd}\,;\ p = \langle b\rangle \bullet ,\ q = \text{‘}d\text{’}\,;\ ]$	Scan (5)
7	$[\, R_{bd}\,;\ p = \langle b\rangle ,\ q = \langle\epsilon\rangle \bullet \text{‘}d\text{’}\,;\ ]$	Complete (6)
8	$[\, R_{bd}\,;\ p = \langle b\rangle ,\ q = \langle d\rangle \bullet\ ;\ ]$	Scan (7)
$8'$	$[\, A\,;\ \Gamma_{b,d}\,]$	
9	$[\, R_g\,;\ p = \langle\epsilon\rangle \bullet \alpha_p,\ q = \alpha_q\,;\ \Gamma_\emptyset,\ \Gamma_\emptyset\,]$	Predict
10	$[\, R_g\,;\ p = \langle a\rangle \bullet A^2.p,\ q = \alpha_q\,;\ \Gamma_{a,c},\ \Gamma_\emptyset\,]$	Combine (9), (4’)
11	$[\, R_g\,;\ p = \langle ab\rangle \bullet ,\ q = \alpha_q\,;\ \Gamma_{a,c},\ \Gamma_{b,d}\,]$	Combine (10), (8’)
12	$[\, R_g\,;\ p = \langle ab\rangle ,\ q = \langle\epsilon\rangle \bullet \alpha_q\,;\ \Gamma_{a,c},\ \Gamma_{b,d}\,]$	Complete (11)
13	$[\, R_g\,;\ p = \langle ab\rangle ,\ q = \langle c\rangle \bullet A^2.q\,;\ \Gamma_{a,c},\ \Gamma_{b,d}\,]$	Combine (12), (4’)
14	$[\, R_g\,;\ p = \langle ab\rangle ,\ q = \langle cd\rangle \bullet\ ;\ \Gamma_{a,c},\ \Gamma_{b,d}\,]$	Combine (13), (8’)
$14'$	$[\, A\,;\ \Gamma_{ab,cd}\,]$	
15	$[\, R_f\,;\ s = \langle\epsilon\rangle \bullet A.p\ A.q\,;\ \Gamma_\emptyset\,]$	Predict
16	$[\, R_f\,;\ s = \langle ab\rangle \bullet A.q\,;\ \Gamma_{ab,cd}\,]$	Combine (15), (4’)
17	$[\, R_f\,;\ s = \langle abcd\rangle \bullet\ ;\ \Gamma_{ab,cd}\,]$	Combine (16), (8’)
$17'$	$[\, S\,;\ \Gamma_w\,]$	

Abbreviations:

$$\begin{aligned} R_f &= S \to f[A] \\ R_g &= A \to g[A^1,\ A^2] \\ R_{ac} &= A \to ac[] \\ R_{bd} &= A \to bd[] \\ \alpha_p &= A^1.p\ A^2.p \\ \alpha_q &= A^1.q\ A^2.q \end{aligned}$$

Figure 4.1: A derivation using the active algorithm.

Earley-style top-down prediction

This is an adaptation of the context-free inference rules 2.4–2.5 in section 2.2.2.

Predict

$$\frac{[\,\dots\,;\,\Gamma,\, r = \rho \bullet A.s\;\alpha,\, \phi\,;\,\dots\,]}{[\,A \to f[\vec{B}]\,;\, r = \rho^{\epsilon} \bullet \alpha,\, \phi\,;\, \vec{\Gamma}_{\emptyset}\,]}\;\left\{\;A \to f[\vec{B}] := r = \alpha,\, \phi\right. \tag{4.14}$$

We only have to add predictions for a category when there already is an item looking for that category.

Initial prediction

$$\frac{}{[\,S \to f[\vec{B}]\,;\, s = \rho^{\epsilon} \bullet \alpha\,;\, \vec{\Gamma}_{\emptyset}\,]}\;\left\{\;S \to f[\vec{B}] := s = \alpha\right. \tag{4.15}$$

We also need initial predictions, for the starting category of the grammar.

Complete, Scan and Combine remain as the inference rules 4.11–4.13.

Example 4.23.

Top-down prediction does not reduce the number of items in figure 4.1; it only specifies the order in which items are predicted. While the predicted items (1), (5), (9) and (15) in the figure can be deduced in any order, top-down prediction specifies that item (15) is inferred by the Initial prediction rule, and the other follows from that by the top-down Predict rule.

However, if the grammar contains more rules, top-down prediction can filter out more items than the basic algorithm.

▲

Kilbury-style bottom-up prediction

This is an adaptation of the context-free inference rule 2.6 in section 2.2.2.

Predict

$$\frac{[\,B_i\,;\,\Gamma_i\,]}{[\,A \to f[\vec{B}]\,;\, r = \rho \bullet \alpha,\, \phi\,;\, \vec{\Gamma}_{\emptyset}[i := \Gamma_i]\,]}\;\left\{\begin{array}{l} A \to f[\vec{B}] := \\ \qquad r = B_i.r'\;\alpha,\, \phi \\ \rho = \Gamma_i.r' \end{array}\right. \tag{4.16}$$

We find the first argument B_i and assure that it has been found previously as the passive item $[\,B_i\,;\,\Gamma_i\,]$. Since we know that the item is found, we implicitly apply the Combine rule and move the dot past the argument.

Terminal

$$\frac{}{[\, A \to f[\,]\,;\, \Gamma \bullet\,;\ \,]} \begin{cases} A \to f[\,] := \phi \\ \Gamma = \langle \phi \rangle \end{cases} \tag{4.17}$$

If the rule does not contain any arguments, it will not be handled by Predict. So we need a new rule for this case.

Complete and Combine remain as the inference rules 4.11 and 4.13.

This version of Kilbury prediction only works for grammars where terminals only occur in rules without arguments, $A \to f[\,]$. All pmcfg grammars can easily be converted to this form as shown by Seki et al. (1991). For grammars in this format, the Scan rule will never apply, so we can safely skip that one.

Another possibility is to augment the Predict rule to handle any grammar rules with terminals; then the Scan rule have to be reintroduced and the Terminal rule can be dropped.

Example 4.24.

Using bottom-up prediction on the example reduces the number of items in figure 4.1 drastically; items (1)–(3), (5)–(7), (9) and (15) will no longer be predicted. Instead the items (4) and (8) will be predicted by the Terminal axiom, and items (10) and (16) will be predicted by the bottom-up Predict rule.

▲

4.5 Parsing of erasing and suppressing PMCFG

A grammar is erasing if some argument projection in some rule does not occur on the right-hand side. This means that some parts of a linearization might not have a realization in the current input string.

Example 4.25.

This is an example of a simple erasing pmcfg;

$$\begin{array}{rcl} S \to f[A] & := & s = A.s_1 \\ A \to g[A,\, B,\, C] & := & s_1 = A.s_2 \,\cdot\, B.s,\ s_2 = A.s_1 \,\cdot\, C.s \\ A \to a[\,] & := & s_1 = \text{`}a_1\text{'},\ s_2 = \text{`}a_2\text{'} \\ B \to b[\,] & := & s = \text{`}b\text{'} \\ C \to c[\,] & := & s = \text{`}c\text{'} \end{array}$$

The grammar recognizes the language,

$$(a_1 \cup a_2 b) \cdot (cb)^* \quad = \quad \{\, a_1,\, a_1cb,\, a_1cbcb,\, \ldots,\, a_2b,\, a_2bcb,\, a_2bcbcb,\, \ldots \}$$

As an example, the string '$a_1 cb$' is recognized by the grammar; but it is impossible to create a corresponding parse item for the A category: $[\, A\,;\ s_1 = 0 \ldots 3,\ s_2 = \mathbf{?}\,]$, since $A.s_2$ then should linearize to '$a_2 bc$'.

▲

One solution is to allow the empty set as a range, meaning that the linearization in question is not found in the input string. The example item will then become $[\,A\,;\,s = 0\ldots3,\,p = \emptyset\,]$. The algorithms do not have to be change at all, and the theoretical space and time complexity does not change either. The problem is that this in practice means that all rules in the grammar will give rise to parse items, even rules where no part of the linearization is recognized. This will yield an extremely big chart, and is therefore impractical.

4.5.1 Removing erasingness from a grammar

A better solution is to translate the grammar into non-erasing form. That this can be done for any PMCFG grammar is already known (Seki et al., 1991). But for completeness, we give an alternative algorithm for removing erasingness from a grammar. The resulting grammar is shown to be a simulation of the original grammar, meaning that it can be directly used for parsing purposes.

We start by defining a relation $\rhd_f$ on projections of categories. The idea is that given a grammar rule $A \to f[\vec{B}] := \phi$, then $A.r \rhd_f B_i.r'$ whenever $B_i.r'$ occurs somewhere in row r of the linearization ϕ.

Definition 4.26. Given a function symbol f with the following rule,

$$A \to f[B_1,\,\ldots,\,B_\delta] \quad := \quad \phi$$

we define a binary relation $\rhd_f$ on projections of categories, as

$$A.r \rhd_f B_i.r' \quad \textbf{iff} \quad \phi.r = \ldots,\,B_i.r',\,\ldots$$

Definition 4.27 (restriction). Given a rule R,

$$A \to f[B_1,\,\ldots,\,B_\delta] \quad := \quad \phi$$

we define the *restriction* $R \mid \Sigma$ by a nonempty set of labels Σ as the new rule,

$$\hat{A} \to \hat{f}[\hat{B}_i \mid \Sigma_i \neq \emptyset] \quad := \quad \{\,(r = \alpha) \in \phi \mid r \in \Sigma\,\}$$

where we by $[\hat{B}_i \mid \Sigma_i \neq \emptyset]$ mean "the sequence of those $\hat{B}_i$ such that Σ_i is nonempty". The new function symbol and the new categories are,

$$\begin{array}{rcl} \hat{f} & = & f[\Sigma] \\ \hat{A} & = & A[\Sigma] \\ \hat{B}_i & = & B_i[\Sigma_i] \end{array}$$

and Σ_i is defined as,

$$\Sigma_i \quad = \quad \{\,r' \mid r \in \Sigma,\,A.r \rhd_f B_i.r'\,\}$$

The linearization ψ of a restriction $R \mid \Sigma$ contains only the rows $r = \alpha$ of the original linearization ϕ, such that $r \in \Sigma$. Furthermore, the categories are restricted to include only the labels that are mentioned in ψ. If Σ_i is the empty set, it means that B_i is not mentioned at all in ψ. Then we have to exclude from the restriction all such categories $\hat{B}_i$, otherwise the restriction will not be a nonerasing rule.

It can be deduced from the definition that the linearization type of a category $\hat{A} = A[\Sigma]$ is a subrecord of the linearization type of A;

$$\hat{A}^\circ \quad = \quad \{\, r : \mathsf{Str} \mid r \in \Sigma \,\} \quad \subseteq \quad A^\circ$$

This also means that if Σ is empty, the linearization type becomes the empty record which is not allowed as a pmcfg linearization type. When creating the nonerasing restriction grammar we have to exclude all categories $\hat{A} = A[\Sigma]$ where Σ is empty.

Algorithm 4.28.

Given a pmcfg grammar G, create a new grammar $\hat{G}$, called the *restriction grammar*, in the following way;

1. Start with $\hat{G}$ containing the restriction $R \mid \Sigma$ for each grammar rule $R = S \to f[\ldots]$, where

$$\Sigma \;=\; S^\circ \;=\; \{\, s : \mathsf{Str} \,\}$$

2. Whenever $\hat{G}$ contains a restriction $R \mid \Sigma$ for the rule $R = A \to f[\ldots B_i \ldots]$, and

$$\Sigma' \quad = \quad \{\, r' \mid r \in \Sigma,\, A.r \rhd_f B_i.r' \,\}$$

is nonempty; add the restriction $R' \mid \Sigma'$ to $\hat{G}$ for each grammar rule $R' = B_i \to g[\ldots]$.

▲

The algorithm terminates since there are only a finite number of restrictions $R \mid \Sigma$ for a given grammar rule R. Note that is possible for a (non-suppressing) erasing grammar rule to loose some of its arguments during conversion. The rule is then called *indirectly suppressing*.

Example 4.29.

The grammar in example 4.25 is erasing. First we calculate the relation $\rhd_{(\cdot)}$,

$$\begin{array}{lcl@{\qquad}lcl}
S.s & \rhd_f & A.s_1 & & & \\
A.s_1 & \rhd_g & A.s_2 & A.s_2 & \rhd_g & A.s_1 \\
A.s_1 & \rhd_g & B.s & A.s_2 & \rhd_g & C.s
\end{array}$$

and from this we see that the g-rule and the a-rule have to be replaced by two restrictions each,

$$\begin{aligned}
\hat{A}_1 \to \hat{g}_1[\hat{A}_2,\, B] &\quad := \quad s_1 = \hat{A}_2.s_2 \;\cdot\; B.s \\
\hat{A}_2 \to \hat{g}_2[\hat{A}_1,\, C] &\quad := \quad s_2 = \hat{A}_1.s_1 \;\cdot\; C.s \\
\hat{A}_1 \to \hat{a}_1[] &\quad := \quad s_1 = \text{`}\mathit{a_1}\text{'} \\
\hat{A}_2 \to \hat{a}_2[] &\quad := \quad s_2 = \text{`}\mathit{a_2}\text{'}
\end{aligned}$$

where $\hat{A}_1 = A[s_1]$ and $\hat{A}_2 = A[s_2]$. Note that both g-restrictions have lost one argument each, so the grammar is indirectly suppressing.

Example 4.30.

The English example grammar from section 1.3.5, as shown in PMCFG format in figure 2.6, is also erasing; the $\mathsf{NP}_{1,2}$ resp. S rules choose only one of the N resp. VP daughter's rows, s_1 or s_2.

Applying the algorithm to this grammar results in a grammar where each noun resp. verb is split into two nouns resp. verbs; a singular and a plural,

$$\begin{aligned}
\widehat{\mathsf{N}}_1 \to \hat{n}_{c1}[] &\;:=\; s_1 = \text{`}\mathit{lion}\text{'} & \qquad \widehat{\mathsf{N}}_2 \to \hat{n}_{c2}[] &\;:=\; s_2 = \text{`}\mathit{lions}\text{'} \\
\widehat{\mathsf{N}}_1 \to \hat{n}_{f1}[] &\;:=\; s_1 = \text{`}\mathit{fish}\text{'} & \qquad \widehat{\mathsf{N}}_2 \to \hat{n}_{f2}[] &\;:=\; s_2 = \text{`}\mathit{fish}\text{'} \\
\widehat{\mathsf{V}}_1 \to \hat{v}_{e1}[] &\;:=\; s_1 = \text{`}\mathit{eats}\text{'} & \qquad \widehat{\mathsf{V}}_2 \to \hat{v}_{e2}[] &\;:=\; s_2 = \text{`}\mathit{eat}\text{'}
\end{aligned}$$

where $\widehat{\mathsf{N}}_i = \mathsf{N}[s_i]$ and $\widehat{\mathsf{V}}_i = \mathsf{V}[s_i]$.

4.5.2 Using the restriction grammar for parsing

To be able to use the restriction grammar $\hat{G}$ for parsing the original grammar, we have to add *metavariables* to parse trees, as defined in section 2.6.1. If the original grammar is suppressing,[3] then for some of its trees, say t, there will be a subtree t' whose linearization will not show up in the linearization of t. This means that t' is exchangeable in t; or in other words, $[\![t]\!] = [\![t[t'/t'']]\!]$ for any t'' of the same category as t'. Since the value of t' is uninteresting as long as it is type-correct, we can use a metavariable **?** in place of t'. In this way we can capture a set of trees all having the same linearization.

Now, since the restricted grammar has removed all (directly or indirectly) suppressing arguments, there will be no representation of the suppressed subtree t' at all in $\hat{G}$. If we are allowed to use metavariables when converting back to trees for the original grammar G, we can add a metavariable whenever necessary.

Lemma 4.31. *The restriction grammar $\hat{G}$ is a trivial simulation of the original grammar G, augmented with metavariables.*

[3] Either directly, or indirectly as in example 4.29.

Proof. Suppose given a tree $g(x_1, \ldots, x_\delta) : \hat{A}$ in $\hat{G}$, where $\hat{A} = A[\Sigma]$, corresponding to the $\hat{G}$-rule,

$$\hat{A} \quad \rightarrow \quad g[X_1, \ldots, X_\delta]$$

Then we know that there is a G-rule,

$$A \quad \rightarrow \quad f[B_1, \ldots, B_\gamma]$$

for which $g = \hat{f}$; and for each X_i there is some B_j such that $X_i = \hat{B}_j = B_j[\Sigma_j]$. From this we can construct the new tree

$$\langle\!| g(x_1, \ldots, x_\delta) |\!\rangle \quad = \quad f(y_1, \ldots, y_\gamma)$$

where $y_j = \langle\!| x_i |\!\rangle$ if $X_i = \hat{B}_j$, and $y_j = \mathbf{?}$ if there is no $X_i = \hat{B}_j$.

Categories and linearizations in $\hat{G}$ are mapped to G by,

$$\langle\!| \hat{A} \,;\, \phi |\!\rangle \quad = \quad A \,;\, \psi$$

where $\hat{A} = A[\Sigma]$, and $\psi = \phi \cup \{\, r = \mathbf{?} \mid r \notin \Sigma \,\}$. □

When constructing trees from the final chart, metavariables can be left unchanged since they represent any possible tree of the correct type.

Example 4.32.

For the restriction grammar in example 4.29, parsing the input string w = '$a_1\, cb$' results in the following pmcfg chart;

1 $[\, S \rightarrow f[\hat{A}_1] \,;\, s = \langle w \rangle \;;\, s_1 = \langle w \rangle \,]$
2 $[\, \hat{A}_1 \rightarrow \hat{g}_1[\hat{A}_2, B] \,;\, s_1 = \langle w \rangle \;;\, s_2 = \langle a_1 c \rangle \;;\, s = \langle b \rangle \,]$
3 $[\, \hat{A}_2 \rightarrow \hat{g}_2[\hat{A}_1, C] \,;\, s_2 = \langle a_1 c \rangle \;;\, s_1 = \langle a_1 \rangle \;;\, s = \langle c \rangle \,]$
4 $[\, \hat{A}_1 \rightarrow \hat{a}_1[] \,;\, s_1 = \langle a_1 \rangle \;;\,]$
5 $[\, B \rightarrow b[] \,;\, s = \langle b \rangle \;;\,]$
6 $[\, C \rightarrow c[] \,;\, s = \langle c \rangle \;;\,]$

Since the simulation is trivial, we can directly convert the chart to a chart for the original grammar in example 4.25, by inserting metavariables whenever necessary;

1	$[\, S \rightarrow f[A] \,;\, \Gamma_1 \,;\, \Gamma_2 \,]$	$\Gamma_1 = \{\, s = \langle w \rangle \,\}$
2	$[\, A \rightarrow g[A, B, C] \,;\, \Gamma_2 \,;\, \Gamma_3, \Gamma_5, \mathbf{?} \,]$	$\Gamma_2 = \{\, s_1 = \langle w \rangle \;;\, s_2 = \mathbf{?} \,\}$
3	$[\, A \rightarrow g[A, B, C] \,;\, \Gamma_3 \,;\, \Gamma_4, \mathbf{?}, \Gamma_6 \,]$	$\Gamma_3 = \{\, s_1 = \mathbf{?} \,;\, s_2 = \langle a_1 c \rangle \,\}$
4	$[\, A \rightarrow a[] \,;\, \Gamma_4 \,;\,]$	$\Gamma_4 = \{\, s_1 = \langle a_1 \rangle \;;\, s_2 = \mathbf{?} \,\}$
5	$[\, B \rightarrow b[] \,;\, \Gamma_5 \,;\,]$	$\Gamma_5 = \{\, s = \langle b \rangle \,\}$
6	$[\, C \rightarrow c[] \,;\, \Gamma_6 \,;\,]$	$\Gamma_6 = \{\, s = \langle c \rangle \,\}$

From this chart we can extract the following only parse tree,

$$f(g(g(a, \mathbf{?}, c), b, \mathbf{?})$$

▲

4.6 Incremental PMCFG parsing

A parsing algorithm is *incremental* if it reads the input one token at the time; and calculates all possible consequences of a token, before the next token is read. This feature is useful for e.g. recognition of spoken input, since language modeling typically requires that probabilities are assigned incrementally. There is also cognitive evidence showing that humans process language in an incremental fashion. For further information about incrementality, see e.g. ACL (2004).

Example 4.33.

The active algorithms in section 4.4 are not incremental. Consider the derivation of the input string '*abcd*' in figure 4.1 and the item (10);

$$10 \qquad [\, A \rightarrow g[A^1,\, A^2]\,;\, p = \langle a \rangle \bullet A^2.p,\; q = A^1.q\; A^2.q\,;\, \Gamma_{a,b},\, \Gamma_{\emptyset}\,]$$

This item is combined with item (8') $[\, A\,;\, \Gamma_{bd}\,]$ into item (11);

$$11 \qquad [\, A \rightarrow g[A^1,\, A^2]\,;\, p = \langle ab \rangle \bullet,\; q = A^1.q\; A^2.q\,;\, \Gamma_{a,c},\, \Gamma_{b,d}\,]$$

But note that this item has only read the first half of the string ($\langle ab \rangle$), while its daughters together have read the full string ($\Gamma_{a,c}$ and $\Gamma_{b,d}$).

A simpler example is when applying the Terminal rule,

$$A \rightarrow ac[] \quad := \quad p = \text{‘}a\text{’},\; q = \text{‘}c\text{’}$$

to the input string '*ca*'; first it will recognize the first row in the range $1 \ldots 2$, and after that it will recognize the second row in the range $0 \ldots 1$.

▲

In this section we describe an incremental, active parsing algorithm. In the end we will see that this algorithm also handles erasing and suppressing PMCFG grammars without modification.

Parse items

We use items of the form,

$$[^k\, A \rightarrow f[\vec{B}]\,;\, \Gamma,\, r = \rho \bullet \alpha,\, \phi\,;\, \vec{\Gamma}\,]$$

where k is an input position; such an item is also called a k-item. The informal meaning is similar to the active items in section 4.4, with the additional constraint that $(j,\, k) \in \rho$ for some j. This means that the item is looking for α starting in position k.

The rows in Γ have been recognized in sequence, which means that the last row in Γ is the latest that has been recognized. Since we cannot know in which order the rows in ϕ will be recognized, we have to treat ϕ as a set of rows, not as a sequence.

Inference rules

Apart from the fact that we have to treat the linearization record as a set instead of a sequence, the basic algorithm is quite similar to the active algorithm in section 4.4. There are four inference rules, and the following shorthands are used;

- By $[^k A\,;\,\Gamma\,]$ we mean any passive item $[^k A \to \ldots\,;\,\Gamma \bullet\,,\,\phi\,;\,\ldots]$. Note that passive items can be unsaturated, meaning that not all rows are recognized (which is true if ϕ is nonempty); contrary to passive items in previous algorithms;
- By $\vec{\Gamma}_\emptyset$ we mean a δ_f-element sequence of empty records;
- By Γ_i we mean the ith element of the sequence $\vec{\Gamma}$; and by $\vec{\Gamma}[i := \Gamma']$ we mean that Γ_i is replaced by Γ'.

PREDICT

$$\frac{}{[^k A \to f[\vec{B}]\,;\, r = \langle\epsilon\rangle \bullet \alpha,\, \phi,\, \psi\,;\, \vec{\Gamma}_\emptyset\,]} \left\{ \begin{array}{l} A \to f[\vec{B}] := \\ \qquad \phi,\, r = \alpha,\, \psi \\ 0 \le k \le |w| \end{array} \right. \tag{4.18}$$

Since we do not know which row will be the first to be recognized, we choose row nondeterministically. Also we do not know from which position k it will be recognized, so this is also nondeterministic.

COMPLETE

$$\frac{[^j R\,;\, \Gamma,\, r = \rho \bullet,\, \phi,\, r' = \alpha,\, \psi\,;\, \vec{\Gamma}\,]}{[^k R\,;\, \Gamma,\, r = \rho,\, r' = \langle\epsilon\rangle \bullet \alpha,\, \phi,\, \psi\,;\, \vec{\Gamma}\,]} \left\{ \; j \le k \le |w| \right. \tag{4.19}$$

Here we also have to choose row and input position k nondeterministically; since the algorithm is incremental, the previous position j has to be less than or equal to k.

SCAN

$$\frac{[^j R\,;\, \Gamma,\, r = \rho \bullet s\; \alpha,\, \phi\,;\, \vec{\Gamma}\,]}{[^k R\,;\, \Gamma,\, r = \rho' \bullet \alpha,\, \phi\,;\, \vec{\Gamma}\,]} \left\{ \begin{array}{l} s = w_{j+1} \ldots w_k \\ \rho' = \rho \cdot \langle s\rangle \end{array} \right. \tag{4.20}$$

When scanning a string s, we have to know that it spans the input positions $j - k$; otherwise the rule is similar to SCAN in section 4.4.

COMBINE

$$\frac{[^j R\,;\, \Gamma,\, r = \rho \bullet B_i.r'\; \alpha,\, \phi\,;\, \vec{\Gamma}\,] \quad [^k B_i\,;\, \Gamma'\,]}{[^k R\,;\, \Gamma,\, r = \rho' \bullet \alpha,\, \phi\,;\, \vec{\Gamma}[i := \Gamma']\,]} \left\{ \begin{array}{l} (j,\, k) \in \Gamma'.r' \\ \rho' = \rho \cdot \Gamma'.r' \\ \Gamma_i \subseteq \Gamma' \end{array} \right. \tag{4.21}$$

The passive B_i-item must have recognized the row r' spanning the positions $j - k$; otherwise the rule is similar to COMBINE in section 4.4.

Example 4.34.

A derivation for the example grammar and the input string '*abcd*' is shown in figure 4.2. Note that the algorithm also predicts a lot of useless items;

- The S-rule R_f introduces one predicted item for each k (5 items);
- Each of the A-rules R_g, R_{ac}, R_{bd} introduces two predicted items for each k, one for each linearization row (30 items);
- Each of the k-items (5), (7) and (8) gives rise to one completed item for each $k' \geq k$ (10 items).

All in all 45 predicted and completed items, of which only 7 are used in the derivation.

▲

4.6.1 Alternative strategies

The inference rules Predict and Complete are extremely crude, predicting any possible row to the right of the dot, anywhere in the input. Obviously this gives rise to several useless items, which shouldn't be there in the first place. Therefore it becomes necessary to have either top-down or bottom-up filtering in the predictions.

Earley-style top-down filtering

The idea with top-down filtering is that we only predict a k-item for $A.r$ if there already is a k-item looking for $A.r$. This can be applied to the Complete rule too; and we finally have to give an initial prediction of the starting category. We write $[^k \bullet A.r\,]$ for a *predict item*, i.e. an item of the form,

$$[^k \ldots ;\, \ldots,\, r' = \rho \bullet A.r \ldots,\, \ldots ;\, \ldots]$$

Predict

$$\frac{[^k \bullet A.r\,]}{[^k A \to f[\vec{B}]\,;\, r = \langle\epsilon\rangle \bullet \alpha,\, \phi,\, \psi\,;\, \vec{\Gamma}_\emptyset\,]} \left\{ \begin{array}{l} A \to f[\vec{B}] := \\ \qquad \phi,\, r = \alpha,\, \psi \end{array}\right. \tag{4.22}$$

Prediction is much more deterministic than in the basic algorithm, since there has to be a predict item in position k already looking for row r.

Complete

$$\frac{[^j R\,;\, \Gamma,\, r = \rho \bullet,\, \phi,\, r' = \alpha,\, \psi\,;\, \vec{\Gamma}\,] \quad [^k \bullet A.r\,]}{[^k R\,;\, \Gamma,\, r = \rho,\, r' = \langle\epsilon\rangle \bullet \alpha,\, \phi,\, \psi\,;\, \vec{\Gamma}\,]} \left\{ \; j \leq k \right. \tag{4.23}$$

Completion is also more deterministic, by the same argument.

1	$[^0\, R_f\,;\ s = \langle\epsilon\rangle \bullet A.p\ A.q\,;\ \Gamma_\emptyset\,]$	Predict
2	$[^0\, R_g\,;\ p = \langle\epsilon\rangle \bullet \alpha_p,\ q = \alpha_q\,;\ \Gamma_\emptyset,\ \Gamma_\emptyset\,]$	Predict
3	$[^0\, R_{ac}\,;\ p = \langle\epsilon\rangle \bullet \text{‘}a\text{’},\ q = \text{‘}c\text{’}\,;\ \,]$	Predict
4	$[^1\, R_{bd}\,;\ p = \langle\epsilon\rangle \bullet \text{‘}b\text{’},\ q = \text{‘}d\text{’}\,;\ \,]$	Predict
5	$[^1\, R_{ac}\,;\ p = \langle a\rangle \bullet,\ q = \text{‘}c\text{’}\,;\ \,]$	Scan (1)
6	$[^1\, R_g\,;\ p = \langle a\rangle \bullet A^2.p,\ q = \alpha_q\,;\ \Gamma_a,\ \Gamma_\emptyset\,]$	Combine (2), (5)
7	$[^2\, R_{bd}\,;\ p = \langle b\rangle \bullet,\ q = \text{‘}d\text{’}\,;\ \,]$	Scan (4)
8	$[^2\, R_g\,;\ p = \langle ab\rangle \bullet,\ q = \alpha_q\,;\ \Gamma_a,\ \Gamma_b\,]$	Combine (6), (7)
9	$[^2\, R_f\,;\ s = \langle ab\rangle \bullet A.q\,;\ \Gamma_{ab}\,]$	Combine (1), (8)
10	$[^2\, R_{ac}\,;\ p = \langle a\rangle,\ q = \langle\epsilon\rangle \bullet \text{‘}c\text{’}\,;\ \,]$	Complete (5)
11	$[^2\, R_g\,;\ p = \langle ab\rangle,\ q = \langle\epsilon\rangle \bullet \alpha_q\,;\ \Gamma_a,\ \Gamma_b\,]$	Complete (8)
12	$[^3\, R_{ac}\,;\ p = \langle a\rangle,\ q = \langle c\rangle \bullet\,;\ \,]$	Scan (10)
13	$[^3\, R_g\,;\ p = \langle ab\rangle,\ q = \langle c\rangle \bullet A^2.q\,;\ \Gamma_{a,c},\ \Gamma_b\,]$	Combine (11), (12)
14	$[^3\, R_{bd}\,;\ p = \langle b\rangle,\ q = \langle\epsilon\rangle \bullet \text{‘}d\text{’}\,;\ \,]$	Complete (7)
15	$[^4\, R_{bd}\,;\ p = \langle b\rangle,\ q = \langle d\rangle \bullet\,;\ \,]$	Scan (14)
16	$[^4\, R_g\,;\ p = \langle ab\rangle,\ q = \langle cd\rangle \bullet;\ \Gamma_{a,c},\ \Gamma_{b,d}\,]$	Combine (13), (15)
17	$[^4\, R_f\,;\ s = \langle abcd\rangle \bullet\,;\ \Gamma_{ab,cd}\,]$	Combine (9), (16)

Abbreviations:

$$\begin{aligned}
R_f &= S \rightarrow f[A] \\
R_g &= A \rightarrow g[A^1,\ A^2] \\
R_{ac} &= A \rightarrow ac[] \\
R_{bd} &= A \rightarrow bd[] \\
\alpha_p &= A^1.p\ A^2.p \\
\alpha_q &= A^1.q\ A^2.q
\end{aligned}$$

Figure 4.2: A derivation using the incremental algorithm.

Initial prediction

$$\frac{}{[^0\, S \to f[\vec{B}]\, ;\, s = \langle\epsilon\rangle \bullet \alpha\, ;\, \vec{\Gamma}_\emptyset\,]} \left\{\; S \to f[\vec{B}] := s = \alpha \right. \tag{4.24}$$

This rule is needed to start the prediction process; we look for the sequence α starting in position 0.

Scan and Combine remain as the inference rules 4.20 and 4.21.

Example 4.35.

The example derivation in figure 4.2 remains exactly the same. The main difference is that this algorithm does not predict as many useless items;

- The items (1) and (6) introduce predictions for $A.p$ at $k = 0, 1$ (6 items);
- The items (9) and (13) introduce predictions for $A.q$ at $k = 2, 3$ (6 items);
- The items (5), (7) and (8) introduce completions for $A.q$ at $k = 2, 3$ (6 items).

In total 19 predicted and completed items (included the initial prediction); as opposed to 45 items in example 4.34.

▲

Kilbury-style bottom-up prediction

The main idea with Kilbury prediction is that we only predict a row if the first thing to look for is already found. And if the thing is found, we can also move the dot forward. Since there are two different rules for predicting (Predict and Complete), and the thing to look for can either be a terminal or an argument, we get four combinations.

Predict+Scan

$$\frac{}{[^k\, A \to f[\vec{B}]\, ;\, r = \langle s\rangle \bullet \alpha,\, \phi,\, \psi\, ;\, \vec{\Gamma}_\emptyset\,]} \left\{ \begin{array}{l} A \to f[\vec{B}] := \\ \qquad \phi,\, r = s\ \alpha,\, \psi \\ s = w_{j+1} \dots w_k \end{array} \right. \tag{4.25}$$

If the row r starts with some terminals occurring in the input string, predict that row and move the dot past the already read terminals.

Predict+Combine

$$\frac{[^k\, B_i\, ;\, \Gamma'\,]}{[^k\, A \to f[\vec{B}]\, ;\, r = \rho \bullet \alpha,\, \phi,\, \psi\, ;\, \vec{\Gamma}_\emptyset[i := \Gamma']\,]} \left\{ \begin{array}{l} A \to f[\vec{B}] := \\ \qquad \phi,\, r = B_i.r'\ \alpha,\, \psi \\ (j,\, k) \in \rho = \Gamma'.r' \end{array} \right. \tag{4.26}$$

If the row r starts with $B_i.r'$, and $B_i.r'$ have been found ending in k, then we can predict the row r and move the dot past $B_i.r'$. When moving the dot forward, we also have to update argument number i to Γ'.

Complete+Scan

$$\frac{[^{j_0}\, R\,;\, \Gamma \bullet\,,\, \phi,\, r = s\; \alpha,\, \psi\,;\, \vec{\Gamma}\,]}{[^{k}\, R\,;\, \Gamma,\, r = \langle s\rangle \bullet \alpha,\, \phi,\, \psi\,;\, \vec{\Gamma}\,]} \quad \begin{cases} s = w_{j+1} \ldots w_k \\ j_0 \leq j \end{cases} \tag{4.27}$$

The same argument as for Predict+Scan, with the added constraint that the terminals should come after the item's previous position j_0.

Complete+Combine

$$\frac{[^{j_0}\, R\,;\, \Gamma \bullet\,,\, \phi,\, r = B_i.r'\; \alpha,\, \psi\,;\, \vec{\Gamma}\,] \quad [^{k}\, B_i\,;\, \Gamma'\,]}{[^{k}\, R\,;\, \Gamma,\, r = \rho \bullet \alpha,\, \phi,\, \psi\,;\, \vec{\Gamma}[i := \Gamma']\,]} \quad \begin{cases} (j,\, k) \in \rho = \Gamma'.r' \\ j_0 \leq j \\ \Gamma_i \subseteq \Gamma' \end{cases} \tag{4.28}$$

The same argument as for Predict+Combine; but the recognized row $B_i.r'$ has to come after the previous position j_0.

Scan and Combine remain as the inference rules 4.20 and 4.21.

This version of the Kilbury algorithm does not work for grammars with ϵ-linearizations. All pmcfg grammars can be converted to ϵ-free form, see Seki et al. (1991) for details. An alternative to removing ϵ-linearizations is to add extra inference rules.

Example 4.36.

First we note that the example grammar is does not have ϵ-linearizations, so the Kilbury algorithm can be used right away.

The derivation in figure 4.2 still basically holds. The only real difference is that all predicted and completed items, (1)–(4), (10), (11) and (14), disappear since they are combined with the following item instead. Also, the rules used to infer the items (5)–(7), (9), (12), (13) and (15) will be one of the four Kilbury rules above.

The main difference is as for the top-down algorithm, the useless predicted and completed items;

- The rules R_{ac}, R_{bd} introduce items by Predict+Scan and Complete+Scan at the positions of the corresponding input tokens (6 items);
- For each of these predicted or completed items, Predict+Combine applies for the rules R_f, R_g (12 items);
- The inference rule Complete+Combine only applies once, yielding item (13) in the derivation (1 item).

All in all 19 predicted or completed items, as compared to the 45 items in example 4.34.

▲

4.6.2 Erasing and suppressing grammars

The incremental algorithm also handles erasing grammars, and even totally suppressed arguments. The problem is how to reconstruct the parse trees from the chart. This can be done by using metavariables as described in section 2.6.1.

Suppressed arguments in an item will show up as $\{\,\}$ in the children's list. All these can simply be seen as metavariables for later purposes, e.g. when building parse trees.

Example 4.37.

We give a derivation for the erasing grammar in example 4.25 and the input string $w =$ '$a_1\,cb$', using the basic incremental algorithm,

1	$[^0\, R_f\ ;\ s = \langle\epsilon\rangle \bullet A.s_1\ ;\ \Gamma_\emptyset\,]$	PREDICT
2	$[^0\, R_g]\ ;\ s_1 = \langle\epsilon\rangle \bullet \alpha_1,\ s_2 = \alpha_2\ ;\ \Gamma_\emptyset,\ \Gamma_\emptyset,\ \Gamma_\emptyset\,]$	PREDICT
3	$[^0\, R_g\ ;\ s_2 = \langle\epsilon\rangle \bullet \alpha_2,\ s_1 = \alpha_1\ ;\ \Gamma_\emptyset,\ \Gamma_\emptyset,\ \Gamma_\emptyset\,]$	PREDICT
4	$[^0\, R_a\ ;\ s_1 = \langle\epsilon\rangle \bullet$ 'a_1', $s_2 =$ 'a_2'; $]$	PREDICT
5	$[^1\, R_c\ ;\ s = \langle\epsilon\rangle \bullet$ 'c'; $]$	PREDICT
6	$[^1\, R_a\ ;\ s_1 = \langle a_1\rangle \bullet ,\ s_2 =$ 'a_2'; $]$	SCAN (4)
7	$[^1\, R_g\ ;\ s_2 = \langle a_1\rangle \bullet C.s,\ s_1 = \alpha_1\ ;\ \Gamma_{a_1},\ \Gamma_\emptyset,\ \Gamma_\emptyset\,]$	COMBINE (3), (6)
8	$[^2\, R_b\ ;\ s = \langle\epsilon\rangle \bullet$ 'b'; $]$	PREDICT
9	$[^2\, R_c\ ;\ s = \langle c\rangle \bullet\ ;\ ]$	SCAN (5)
10	$[^2\, R_g\ ;\ s_2 = \langle a_1 c\rangle \bullet ,\ s_1 = \alpha_1\ ;\ \Gamma_{a_1},\ \Gamma_\emptyset,\ \Gamma_c\,]$	COMBINE (7), (9)
11	$[^2\, R_g\ ;\ s_1 = \langle a_1 c\rangle \bullet B.s,\ s_2 = \alpha_2\ ;\ \Gamma_{a_1 c},\ \Gamma_\emptyset,\ \Gamma_\emptyset\,]$	COMBINE (2), (10)
12	$[^3\, R_b\ ;\ s = \langle b\rangle \bullet\ ;\ ]$	SCAN (8)
13	$[^3\, R_g\ ;\ s_1 = \langle a_1 cb\rangle \bullet ,\ s_2 = \alpha_2\ ;\ \Gamma_{a_1 c},\ \Gamma_b,\ \Gamma_\emptyset\,]$	COMBINE (11), (12)
14	$[^3\, R_f\ ;\ s = \langle a_1 cb\rangle \bullet A.s_1\ ;\ \Gamma_{a_1 cb}\,]$	COMBINE (1), (13)

where we use the following abbreviations;

R_f	$=$	$S \to f[A]$	Γ_{a_1}	$=$	$\{\, s_1 = \langle a_1\rangle \,\}$
R_g	$=$	$A \to g[A,\ B,\ C]$	Γ_b	$=$	$\{\, s = \langle b\rangle \,\}$
R_a	$=$	$A \to a[]$	Γ_c	$=$	$\{\, s = \langle c\rangle \,\}$
R_b	$=$	$B \to b[]$	$\Gamma_{a_1 c}$	$=$	$\{\, s_2 = \langle a_1 c\rangle \,\}$
R_c	$=$	$C \to c[]$	$\Gamma_{a_1 cb}$	$=$	$\{\, s_1 = \langle a_1 cb\rangle \,\}$
α_1	$=$	$A.s_2\ B.s$	α_2	$=$	$A.s_1\ C.s$

Note that the passive items (6), (9), (10), (12), (13) and (14) correspond to the items (4), (6), (3), (5), (2) and (1) respectively in example 4.32. Also note that the basic incremental algorithm predict several useless items, which are not noted in the derivation.

▲

4.7 Summary

In this chapter we defined four different tabular parsing algorithms for context-free GF and PMCFG. First we gave a general passive algorithm, which works for context-free GF grammars. Then we showed how to use a context-free approximation for PMCFG parsing; the PMCFG grammar is converted to an over-generating CFG, which is used for parsing. Afterwards the resulting context-free chart is converted back to a PMCFG chart, from which unsound items have to be removed, since the CFG is over-generating.

Finally we gave two active parsing algorithms for PMCFG; the first is a basic algorithm which recognizes the linearization rows of a rule in a fixed order. The second algorithm recognizes rows incrementally according to the order in which they occur in the input. Both top-down and bottom-up prediction strategies were investigated.

It is only the last, incremental algorithm that can handle erasing and suppressing PMCFG grammars without modification. For the other three algorithms, we gave an algorithm for removing erasingness from a grammar; we also showed that the resulting nonerasing grammar is a simulation and thus can be used for parsing the original grammar.

Chapter 5

Extensions of concrete syntax

This chapter describes three possible extensions of GF, context-free GF and PMCFG, one of which has two different possible interpretations. Apart from investigating the resulting expressive power and parsing complexity, we also give active parsing algorithms for each of the extensions.

The intersection operation, borrowed from *conjunctive grammar* (Okhotin, 2001), make PMCFG equivalent to *simple literal movement grammar* (Groenink, 1997a,b) and *range concatenation grammar* (Boullier, 2000a,b). As a corollary we get that conjunctive PMCFG describe exactly the class of languages recognizable in polynomial time.

The disjunction operation can have two possible interpretations; one intensional which does not change the descriptive power of context-free GF and PMCFG, and one extensional which is conjectured to be a strict extension. With extensional disjunction it is possible to describe the language $(a \cup b)^{2^n}$, which is conjectured cannot be described by context-free GF and PMCFG.

The third operation is the interleaving operation, which is borrowed from *partially ordered multiset context-free grammar* (*poms*-CFG; Nederhof et al., 2003) which in turn is a variant of the ID/LP formalism (Shieber, 1984). This operation can be reduced to a number of disjunctions, but this reduction can lead to an exponential increase of the grammar size. We instead give a direct parsing algorithm derived from a parsing algorithm for *poms*-CFG.

A note on PMCFG vs. context-free GF

In this chapter we write "GF/PMCFG" when the surrounding text applies both for full GF and PMCFG. When the context applies for context-free GF and PMCFG, we use only the term "PMCFG", since the two formalisms are equivalent as shown in chapter 3.

5.1 Intersection (&)

There is an extension of context-free grammars called CONJUNCTIVE GRAMMAR, introduced by Okhotin (2001), where the right-hand sides of rules are extended with a new intersection operator. A conjunctive context-free rule is written,

$$A \quad \rightarrow \quad \alpha_1 \,\&\, \ldots \,\&\, \alpha_n$$

where $\alpha_i \in (N \cup \Sigma)^*$. The interpretation is that A can be rewritten to $w \in \Sigma^*$ iff all α_i can be rewritten to w. This operation can be directly transferred to GF/PMCFG linearizations.

Definition 5.1 (intersection). The intersection operation is a partial linearization operation with the following definition; $\phi_1 \,\&\, \phi_2$ is calculated to ϕ_1 iff $\phi_1 = \phi_2$.

We call GF/PMCFG extended with the intersection operation *conjunctive* GF/PMCFG. The following laws hold for intersections of linearizations:

$$\begin{array}{rcl}
\phi \,\&\, \phi & = & \phi \\
\alpha\ (\beta_1 \,\&\, \beta_2)\ \gamma & = & (\alpha\ \beta_1\ \gamma) \,\&\, (\alpha\ \beta_2\ \gamma) \\
\phi,\ r = \alpha_1 \,\&\, \alpha_2,\ \psi & = & (\phi,\ r = \alpha_1,\ \psi) \,\&\, (\phi,\ r = \alpha_2,\ \psi)
\end{array}$$

This means that we can push out an intersection to a row, which is used in the active parsing algorithm described in section 5.1.3. We can even push out an intersection to an intersection of linearizations.

Example 5.2.

In the end of section 1.3.5, we introduced discontinuous verb phrases (with the rows s_1, s_2) to handle some phenomena in Swedish syntax. Even English syntax needs discontinuous verb phrases to handle e.g. topicalization as in '*it is fish that many lions eat*'.

Groenink (1997a,b) suggests to handle verb phrase coordination by using conjunction on the verb component of the verb phrase. In PMCFG format, this looks like follows;

$$\begin{array}{rcl}
\mathsf{VP} \rightarrow vp_c[\mathsf{VP}^1,\ \mathsf{VP}^2] & := & s_1 = \mathsf{VP}^1.s_1\ \text{'}\mathit{and}\text{'}\ \mathsf{VP}^2.s_1, \\
 & & s_2 = \mathsf{VP}^1.s_2 \,\&\, \mathsf{VP}^2.s_2
\end{array}$$

By combining two verb phrases with the same object, we can form a coordinated verb phrase;

$$\begin{aligned} vp^{\circ}_{c}(\{\, s_1 = \text{'}\textit{catch}\text{'},\, s_2 = \text{'}\textit{fish}\text{'}\, \}, & \\ \{\, s_1 = \text{'}\textit{eat}\text{'},\, s_2 = \text{'}\textit{fish}\text{'}\, \}) \quad & = \quad \{\, s_1 = \text{'}\textit{catch and eat}\text{'},\, s_2 = \text{'}\textit{fish}\text{'}\, \} \end{aligned}$$

which in turn can be used to form sentences like '*many lions catch and eat fish*', or the topicalized version '*it is fish that many lions catch and eat*'.

▲

5.1.1 A strict extension

Theorem 5.3. *The class of languages recognized by conjunctive* GF/PMCFG *grammars is closed under intersection.*

Proof. Let G_1 and G_2 be two grammars (with no common categories or function symbols) recognizing the languages $\mathcal{L}(G_1)$ and $\mathcal{L}(G_2)$ respectively. Let G contain all rules from G_1 and G_2 plus the following single rule for the new starting category S:

$$S \to f[S_1,\, S_2] := s = S_1.s \,\&\, S_2.s$$

It is trivial to see that G recognizes all and only those strings that are recognized by both G_1 and G_2.

□

Corollary 5.4. *The intersection operation is a strict extension of* PMCFG.

The corollary follows from the fact that PMCFG is not closed under intersection Seki et al. (1991), a property it shares with context-free grammars.

Language-theoretic implications

Closedness under intersection has some less desirable properties, which conjunctive PMCFG inherits from CONJUNCTIVE GRAMMAR (Okhotin, 2001).

- The following decision problems are undecidable: emptiness, finiteness, regularity, context-freeness, inclusion and equivalence. This is because these decision problems are undecidable for finite intersections of context-free grammars, see e.g. Hopcroft and Ullman (1979).
- Conjunctive PMCFG is not closed under homomorphism. This follows from the fact that any recursively enumerable language $\mathcal{L}$ can be described by $h(\mathcal{L}_1 \cap \mathcal{L}_2)$, for some homomorphism h and context-free languages $\mathcal{L}_1$, $\mathcal{L}_2$, see e.g. Ginsburg (1975).

Usefulness of intersection

Conjunctive GF/PMCFG is not only closed under intersection, but the closure is also *modular*, i.e. it preserves the structure of the underlying grammar conjuncts. This makes it useful for modular grammar engineering, as noted by Boullier (2000a,b). Intersection might also be useful for modeling secondary/tertiary structures of biological sequences, as has been investigated by Chiang (2004).

For purely linguistic phenomena, Groenink (1997a) has a suggestion of how to use intersection to describe verb coordination, as shown in example 5.2.

5.1.2 Conjunctive PMCFG describes the polynomial languages

In this section we show that conjunctive PMCFG is equivalent to the formalisms s-LMG and RCG. Since it is already known that these formalisms exactly describe the class of languages recognizable in polynomial time, we get the same result for PMCFG extended with a intersection operation.

Literal movement grammar and range concatenation grammar

Literal movement grammar (LMG; Groenink, 1997a,b), and its relative range concatenation grammar (RCG; Boullier, 2000a,b), are grammar formalisms based on *predicates* over string tuples. A grammar is a collection of *clauses* for predicates, very similar to the programming language PROLOG. We here define the general formalism of LMG, and then two equivalent subclasses, RCG and *simple* LMG. We assume given a finite set Σ of terminal tokens, and an infinite supply of logical variables $x_1, x_2, \ldots \in \mathsf{Var}$.

Definition 5.5 (clause, predicate). A *clause* is of the form $\phi \vdash \psi_1, \ldots, \psi_m$ where each of $\phi, \psi_1, \ldots, \psi_m$ are predicates. A *predicate* is a term $A(\alpha_1, \ldots, \alpha_n)$, where each $\alpha_i \in (\Sigma \cup \mathsf{Var})^*$ is a concatenative sequence of terminals and logical variables. A clause can be *instantiated* by substituting a string for each variable in the clause.

A literal movement grammar is a finite number of clauses together with a designated start predicate. To define the language of a LMG grammar G, we define a rewriting relation $\Rightarrow_G$ on sequences of instantiated predicates,

$$\Gamma_1, \bar{\phi}, \Gamma_2 \quad \Rightarrow_G \quad \Gamma_1, \bar{\psi}_1, \ldots, \bar{\psi}_m, \Gamma_2$$

whenever $\bar{\phi} \vdash \bar{\psi}_1, \ldots, \bar{\psi}_m$ is an instantiation of a clause in G. The language of a grammar is then $\mathcal{L}(G) = \{ w \in \Sigma^* \mid S(w) \Rightarrow_G^* \epsilon \}$, where S is the start predicate in G.

LMG is a very general, Turing-complete, grammar formalism. To get a recognizable subclass of LMG, one can consider two possibilities; to restrict the definition of clause instantiation, or to put syntactic restrictions on the form of the predicates.

Definition 5.6 (RCG). A range concatenation grammar (rcg) is an lmg with a restricted form of clause instantiation. A clause can only be instantiated by substrings of the given input string; i.e. if $\phi \vdash \psi_1, \ldots, \psi_m$ is an instantiation of a clause, then all arguments to $\phi, \psi_1, \ldots, \psi_m$ are substrings of the input. This has the effect that all strings in a rcg can be replaced by pairs of input positions, called ranges, as explained in section 4.1.[1]

As an example, if the input string is '$b\ a\ c\ h$', then for the following clauses,

$$\begin{array}{rcl} A(bac) & \vdash & B(b), C(c) \\ A(bach) & \vdash & B(b), C(ch) \\ A(back) & \vdash & B(b), C(ck) \end{array}$$

the first two are rcg instantiations of the clause $A(x\ a\ z) \vdash B(x), C(z)$; but not the third.

Definition 5.7 (s-LMG). A *simple* lmg (s-lmg) is an lmg where each clause $\phi \vdash \psi_1, \ldots, \psi_m$ obeys the following restrictions:

- **Non-combinatorial (NC):** The arguments of the right-hand side predicates are variables;
- **Bottom-up nonerasing (BNE):** Each variable in the right-hand side also occurs in the left-hand side;
- **Bottom-up linear (BL):** No variable occurs more than once in the left-hand side.

Both these formalisms are equivalent, since they describe exactly the class of languages recognizable in polynomial time (Groenink, 1997b,a; Boullier, 2000a,b; Bertsch and Nederhof, 2001). Note that s-lmg/rcg are closed under intersection; if S_1 and S_2 are the start predicates of G_1 and G_2, then $S(x) \vdash S_1(x), S_2(x)$ defines the intersection of the languages $\mathcal{L}(G_1)$ and $\mathcal{L}(G_2)$.

There is an alternative formulation of s-lmg; we can remove the restriction on bottom-up linearity and instead add top-down nonerasingness:

- **Top-down nonerasing (TNE):** Each variable in the left-hand side also occurs in the right-hand side.

The following lemma states that TNE and BL are equivalent restrictions in the context of NC and BNE; i.e. that either of TNE and BL can be used when defining s-lmg.

[1] Boullier (2000a,b) defines rcg predicates directly on ranges, but this definition is equivalent.

Lemma 5.8. *TNE and BL are equivalent in the following sense;*

1. *Any* LMG *clause* $\phi \vdash \psi_1, \ldots, \psi_m$ *can be converted to an equivalent top-down nonerasing (TNE) clause;*
2. *Any* LMG *clause can be converted to an equivalent bottom-up linear (BL) clause;*
3. *Both conversions preserve NC and BNE.*

Proof.

1. Assume that there is a variable x in ϕ not occurring in any of $\psi_1, \ldots, \psi_m$. Add the predicate call $\mathsf{Str}(x)$ to the right-hand side, with the definition,

$$\begin{array}{rcll} \mathsf{Str}(\epsilon) & \vdash & \epsilon & \\ \mathsf{Str}(s\ x) & \vdash & \mathsf{Str}(x) & \quad \text{(for each } s \in \Sigma\text{)} \end{array}$$

   This new clause is equivalent, since the predicate $\mathsf{Str}(x)$ only says that x is a string.

2. (Groenink, 1997a,b) Assume that there is a variable x occurring twice in ϕ. Replace one occurrence by a new variable x', and add the predicate call $\mathsf{Eq}(x, x')$ to the right-hand side, with the definition,

$$\begin{array}{rcll} \mathsf{Eq}(\epsilon, \epsilon) & \vdash & \epsilon & \\ \mathsf{Eq}(s\ x, s\ y) & \vdash & Eq(x, y) & \quad \text{(for each } s \in \Sigma\text{)} \end{array}$$

   This new clause is equivalent, since the predicate $\mathsf{Eq}(x, y)$ says that the two arguments are equal strings.

3. The conversions preserve NC, since the predicates $\mathsf{Str}(x)$ and $\mathsf{Eq}(x, x')$ are non-combinatorial. Furthermore, they preserve BNE, since the only variable that is introduced on the left-hand side (x') is also introduced on the right-hand side.

□

In the following we will use the alternative definition of s-LMG; where clauses are NC, BNE and TNE.

Equivalence of PMCFG and s-LMG/RCG

Here we use the original definition of PMCFG rules; as functions over string tuples, not records. In this setting, a PMCFG rule looks like,

$$\begin{array}{rcl} A & \rightarrow & f[B_1, \ldots, B_\delta] \\ f^\circ(x_{1,1}, \ldots, x_{1,n_1}; & & \\ \ldots; & & \\ x_{\delta,1}, \ldots, x_{\delta,n_\delta}) & = & \alpha_1, \ldots, \alpha_n \end{array}$$

where each α_i is a sequence of strings and bound variables. We also assume that the linearizations are nonerasing, i.e. that each variable $x_{i,j}$ occurs in some α_k; recall from section 2.5.3 that this is not a real restriction on the expressivity of PMCFG.

It is straightforward to convert a nonerasing PMCFG grammar into an equivalent s-LMG grammar. Each PMCFG rule above is converted to the equivalent s-LMG clause,

$$\begin{array}{rcl} A(\alpha_1, \ldots, \alpha_n) & \vdash & B_1(x_{1,1}, \ldots, x_{1,n_1}), \\ & & \ldots, \\ & & B_\delta(x_{\delta,1}, \ldots, x_{\delta,n_\delta}) \end{array}$$

Note that this clause is NC (since each of the $x_{i,j}$ is a variable), BNE (since f° is nonerasing) and TNE (since f° is a function), and therefore the clause is s-lmg.

Lemma 5.9. *Any conjunctive* PMCFG *can be converted to an equivalent* s-LMG*.*

PROOF. Since intersections can be pushed out, we can assume that the PMCFG rules are of the form,

$$\begin{array}{rcl} A & \to & f[B_1, \ldots, B_\delta] \\ f^\circ(x_{1,1}, \ldots, x_{1,n_1}\,; & & \\ \ldots\,; & & \\ x_{\delta,1}, \ldots, x_{\delta,n_\delta}) & = & \alpha_{1,1} \,\&\, \ldots \,\&\, \alpha_{1,c_1}, \\ & & \ldots, \\ & & \alpha_{n,1} \,\&\, \ldots \,\&\, \alpha_{n,c_n} \end{array}$$

where each $\alpha_{i,j}$ is a sequence of strings and variables, as above. Translate this to the s-LMG clause,

$$\begin{array}{rcl} \hat{A}(\alpha_{1,1} \,\&\, \ldots \,\&\, \alpha_{1,c_1}\,; & & \\ \ldots\,; & & \\ \alpha_{n,1} \,\&\, \ldots \,\&\, \alpha_{n,c_n}) & \vdash & B_1(x_{1,1}, \ldots, x_{1,n_1}), \\ & & \ldots, \\ & & B_\delta(x_{\delta,1}, \ldots, x_{\delta,n_\delta}) \end{array}$$

where the left-hand side is just syntactic sugar for a predicate with arity $c_1 + \cdots + c_n$. If the PMCFG rule is nonlinear, we can utilize the same transformation as in the proof of lemma 5.8, by adding calls to $Eq(x, y)$. Finally, add coercion clauses for $\hat{A}(\ldots)$, implementing the intersections,

$$A(x_1, \ldots, x_n) \;\vdash\; \hat{A}(x_1 \,\&\, \ldots \,\&\, x_1\,; \ldots ;\, x_n \,\&\, \ldots \,\&\, x_n)$$

The resulting s-LMG grammar is equivalent to the PMCFG grammar. □

Lemma 5.10. *Any s-LMG can be converted to an equivalent conjunctive* PMCFG.

Proof. A s-LMG predicate is of the form,

$$\begin{aligned} A(\alpha_1, \ldots, \alpha_n) \ \vdash \ & B_1(x_{1,1}, \ldots, x_{1,n_1}), \\ & \ldots, \\ & B_\delta(x_{\delta,1}, \ldots, x_{\delta,n_\delta}) \end{aligned}$$

If the variables $x_{i,j}$ all are distinct, it is equivalent to the PMCFG rule,

$$\begin{aligned} A \ &\rightarrow \ f[B_1, \ldots, B_\delta] \\ f^\circ(x_{1,1}, \ldots, x_{1,n_1}\,; & \\ \ldots\,; & \\ x_{\delta,1}, \ldots, x_{\delta,n_\delta}) \ &= \ \alpha_1, \ldots, \alpha_n \end{aligned}$$

However, in s-LMG, the variables in the right-hand side of a clause need not be distinct. Assume therefore that $x_{i',j'} = x_{i,j} = x$. Now, introduce a new variable x' to replace x as $x_{i',j'}$; and replace each occurrence of x in the right-hand side with the conjunction $(x \,\&\, x')$. The resulting rule is a correct conjunctive PMCFG rule, and equivalent to the given s-LMG clause. □

Theorem 5.11. *Conjunctive* PMCFG, *s-LMG and* RCG *are equivalent.*

Corollary 5.12. *The class of languages recognizable by conjunctive* PMCFG *is exactly the class of languages recognizable in polynomial time.*

Example 5.13.

The following is the result of translating the PMCFG rule for verb coordination in example 5.2, into s-LMG/RCG;

$$\begin{aligned} \widehat{\mathsf{VP}}(x_1 \text{ `\textit{and}' } y_1\,;\ x_2 \,\&\, y_2) \ &\vdash \ \mathsf{VP}(x_1,\, x_2),\ \mathsf{VP}(y_1,\, y_2) \\ \mathsf{VP}(x,\, y) \ &\vdash \ \widehat{\mathsf{VP}}(x\,;\ y \,\&\, y) \end{aligned}$$

After simplifying away $\widehat{\mathsf{VP}}$, we get the same rule as in Groenink (1997a);

$$\mathsf{VP}(x \text{ `\textit{and}' } y,\ z) \ \vdash \ \mathsf{VP}(x,\, z),\ \mathsf{VP}(x,\, z)$$

▲

5.1.3 Parsing of conjunctive PMCFG

Ranges and intersection

We can use exactly the same definition of ranges as in section 4.1. The general definition of range intersection is set intersection, $\rho_1 \,\&\, \rho_2 = \rho_1 \cap \rho_2$. For string-equivalent ranges this boils down to a simple equality check ($\rho_1 \,\&\, \rho_2$ is calculated to ρ_1 iff $\rho_1 = \rho_2$), since the string-equivalent ranges form a partition of $\mathcal{R}_w$.

The notion of range-restriction can be extended to also include intersection. This is possible since there is a range interpretation of intersection. The following theorem is a direct consequence of the fact that range intersection can be interpreted as range equality for string-equivalent ranges.

Theorem 5.14. *The parsing algorithms for context-free* GF *of section 4.2 has polynomial time complexity for conjunctive* PMCFG.

Active parsing

Here we describe a simple extension of the active parsing algorithm in section 4.4. We assume that an intersection $r = \alpha_1 \,\&\, \ldots \,\&\, \alpha_n$ in a linearization is written as a number of consecutive rows in the record, $r = \alpha_1, \ldots, r = \alpha_n$. This is a slight abuse of notation, but the justification is that the inference rules 4.10–4.13 in section 4.4 only need minimal changes.

Intersect

$$\frac{[\,R\,;\,\Gamma,\, r = \rho_0,\, r = \rho_1 \bullet\,,\, \phi\,;\, \vec{\Gamma}\,]}{[\,R\,;\,\Gamma,\, r = \rho_0 \bullet\,,\, \phi\,;\, \vec{\Gamma}\,]} \;\left\{\; \rho_0 = \rho_1 \right. \tag{5.1}$$

This is the only extra rule, taking care of the intersection of two linearizations; if two linearizations ρ_0 and ρ_1 are found for the same label, they have to be equal.

Complete

$$\frac{[\,R\,;\,\Gamma,\, r = \rho \bullet\,,\, r' = \alpha,\, \phi\,;\, \vec{\Gamma}\,]}{[\,R\,;\,\Gamma,\, r = \rho,\, r' = \langle\epsilon\rangle \bullet \alpha,\, \phi\,;\, \vec{\Gamma}\,]} \;\left\{\; \Gamma \neq (\ldots,\, r = \rho_1) \right. \tag{5.2}$$

The only difference to the original rule is the extra side condition; we have to state that Complete must not apply when Intersect applies.

Predict, Scan and Combine remain as the rules 4.10, 4.12 and 4.13.

5.2 Intensional disjunction (|)

The dual operation of intersection is disjunction, which we write as $\phi_1 \mid \phi_2$. This can also be added to GF/PMCFG linearizations, which we then call *disjunctive* GF/PMCFG.

There are two possible interpretations of the disjunction operation; an intensional and an extensional. The first does not change the descriptive power of PMCFG, but we conjecture that the second does. In this section we describe the intensional, and the next section takes care of the extensional version.

Example 5.15.

The first GF grammar in section 1.3.5 uses disjunctive linearizations for the terms n_c° and v_e°;

$$\begin{aligned} \mathsf{N} \to n_c[] &\;:=\; s = \text{‘}\textit{lion}\text{’} \mid \text{‘}\textit{lions}\text{’} \\ \mathsf{V} \to v_e[] &\;:=\; s = \text{‘}\textit{eats}\text{’} \mid \text{‘}\textit{eat}\text{’} \end{aligned}$$

▲

Example 5.16.

This is another simple grammar using disjunction,

$$\begin{aligned} S \to f[S] &\;:=\; s = S.s\ S.s \\ S \to a[] &\;:=\; s = \text{‘}a\text{’} \mid \text{‘}b\text{’} \end{aligned}$$

and since it also makes use of reduplication, it makes a good example for discussion the differences between intensional and extensional disjunction.

▲

Disjunction as a non-deterministic operation

To be able to define intensional disjunction, we must extend the definition of GCFG to *many-valued*, or non-deterministic, linearization functions, as discussed in section 2.4.2.

Definition 5.17 (intensional disjunction). Intensional disjunction is a non-deterministic operation with the following definition; $\phi_1 \mid \phi_2$ is calculated to either ϕ_1 or ϕ_2.

The following laws hold for intensional disjunctions (as for intersections):

$$\begin{aligned} \alpha\ (\beta_1 \mid \beta_2)\ \gamma &= (\alpha\ \beta_1\ \gamma) \mid (\alpha\ \beta_2\ \gamma) \\ \phi,\ r = \alpha_1 \mid \alpha_2,\ \psi &= (\phi,\ r = \phi_1,\ \psi) \mid (\phi,\ r = \phi_2,\ \Phi') \end{aligned}$$

This means that we can push out a disjunction to a row, which is used in the active parsing algorithm below. We can even push out a disjunction to a disjunction of linearizations, which is used when showing that the extension is not strict.

Example 5.18.

The reduplication grammar in example 5.16 have trees of the form $f^k(a)$; i.e. k applications of f, finally applied to a. There are two possible linearizations of the term a, and each application of f duplicates the string; thus the language is $\mathcal{L}_{\mathsf{int}} = a^{2^n} \cup b^{2^n}$, when we use the intensional semantics. This language is an exponentially growing language, and hence not mildly context-sensitive.

▲

5.2.1 A non-strict extension

To see that the intensional disjunction is not a strict extension, we give a translation from disjunctive GF/PMCFG to ordinary GF/PMCFG.

Since it is possible to push out disjunctions, we can assume that each disjunctive rule is of the form $A \to f[\vec{B}] := \alpha_1 \mid \ldots \mid \alpha_n$, where none of ϕ_i contains a disjunction. Such a rule is translated to n rules $A \to f_i[\vec{B}] := \alpha_i$ for $1 \leq i \leq n$. That the grammars are equivalent comes from the compositionality of GCFG; any rule taking a term of type A as argument cannot separate the functions f_i from each other, thus they are indistinguishable.

5.2.2 Parsing of intensionally disjunctive PMCFG

Here we describe an extension of the active parsing algorithm described in section 4.4, to handle intensional disjunctions. The only difference to the original algorithm is the COMPLETE rule, where we non-deterministically choose any of the disjunctions.

COMPLETE

$$\frac{[\,R;\,\Gamma,\,r=\rho\,\bullet\,,\,r'=\alpha_1 \mid \ldots \mid \alpha_n,\,\phi;\,\vec{\Gamma}\,]}{[\,R;\,\Gamma,\,r=\rho,\,r'=\rho^{\epsilon}\bullet\alpha_i,\,\phi;\,\vec{\Gamma}\,]}\;\Big\{\;\; 1 \leq i \leq n \qquad (5.3)$$

PREDICT, SCAN AND COMBINE remain as the rules 4.10, 4.12 and 4.13.

5.3 Extensional disjunction (|)

To define extensional disjunctions, we do not change the definition of GCFG, but instead we change what is meant by linearization types.

Linearizations as sets

We lift the linearization types to non-empty sets of linearizations. This means that an (extensional) linearization, written Φ, is a set of (original) linearizations, still written ϕ. We have to redefine all existing linearization operations in the following way:

- Concatenation is applied to sets of strings, in the standard manner;

$$\Phi_1 \cdot \Phi_2 \;=\; \{\, \alpha_1 \cdot \alpha_2 \mid \alpha_1 \in \Phi_1,\, \alpha_2 \in \Phi_2 \,\}$$

- Record formation returns a set of records;

$$\{\, r_1 = \Phi_1\,;\; \ldots\,;\; r_n = \Phi_n \,\} \quad = \quad \{\, \{\, r_1 = \phi_1\,;\; \ldots\,;\; r_n = \phi_n \,\} \mid \phi_i \in \Phi_i,\, 1 \leq i \leq n \,\}$$

- Record projection is also lifted to sets of linearizations;

$$\Phi.r \quad = \quad \{\, \phi.r \mid \phi \in \Phi \,\}$$

Disjunction as set union

With these changes we can define linearization of disjunctions as set union

Definition 5.19 (extensional disjunction). Extensional disjunction is an operation on extensional linearizations defined as $\phi_1 \mid \phi_2 = \phi_1 \cup \phi_2$.

Unfortunately, the natural law of η-conversion, $\Phi = \{\, r_1 = \Phi.r_1\,;\; \ldots\,;\; r_n = \Phi.r_n \,\}$, does not hold any more. Instead we have the much weaker law,

$$\Phi \quad \subseteq \quad \{\, r_1 = \Phi.r_1\,;\; \ldots\,;\; r_n = \Phi.r_n \,\}$$

The following law for string concatenation still holds though,

$$\alpha\,(\beta_1 \mid \beta_2)\,\gamma \quad = \quad (\alpha\,\beta_1\,\gamma) \mid (\alpha\,\beta_2\,\gamma)$$

So, disjunctions cannot be pushed out to the linearization definitions, and the translation to PMCFG in section 5.2.1 cannot be applied.

5.3.1 A strict extension

The effect of extensional disjunction is to lift reduplication to work on the tree level instead of the string level. This means that each reduplication in a linearization can linearize to different strings, as long as they are represented by the same tree. With extensional disjunction it is possible to define languages not obviously definable by ordinary PMCFG linearizations.

Example 5.20.

The grammar in example 5.16 generates the language $\mathcal{L}_{\mathsf{ext}} = (a \cup b)^{2^n}$ using the extensional semantics. The reason for this is that each application of f duplicates the length of the string, but otherwise only says that the string should consist of a's and b's.

▲

Note that this language is larger than the intensional language for the same grammar, $\mathcal{L}_{\mathsf{int}} \subsetneq \mathcal{L}_{\mathsf{ext}}$. To be precise, very much larger; there are 2 strings in $\mathcal{L}_{\mathsf{int}}$ of length 2^n, while there are 2^{2^n} strings of the same length in $\mathcal{L}_{\mathsf{ext}}$.

Conjecture 5.21. *There is no* PMCFG *grammar that can express the language* $\mathcal{L}_{\mathsf{ext}} = (a \cup b)^{2^n}$.

Unfortunately we have not yet found a proof of the conjecture, but we give an informal argument of why it should be true.

PROOF (IDEA). The argument is based on the fact that each PMCFG tree has exactly one linearization. We argue that $\mathcal{L}_{\mathsf{ext}}$ contains more strings of length 2^n than any candidate PMCFG grammar can have corresponding trees;

- There are $2^{2^{n+1}} = \left(2^{2^n}\right)^2$ legal strings of length 2^{n+1};
- Any PMCFG string of length 2^{n+1} must be composed of strings of length $\leq 2^n$;
- But there are only $\sum_{i=1}^{n} 2^{2^i} < 2\left(2^{2^n}\right) \ll \left(2^{2^n}\right)^2$ legal strings of length $\leq 2^n$;

So, a PMCFG function f creating rules of length 2^{n+1} must take at least two arguments with strings of length $\leq 2^n$. But, there is no way of guaranteeing that two argument strings x and y have equal length. This means that if $2^j = |x| \neq |y| = 2^k$, then $|f^\circ(x, y)| = 2^j + 2^k \neq 2^n$ for any n, and therefore f must accept strings of length $\neq 2^n$. We have a contradiction.

□

The reason why this is not a correct proof is that it does not cover all possible PMCFG grammars. Note that the argument hinges on reduplication; without the possibility of reduplication, extensional and intensional disjunction are equivalent.

Corollary 5.22. *Extensional disjunctive* PMCFG *is a strict extension of* PMCFG *(if the conjecture holds).*

Note however that the argument does not hold for conjunctive PMCFG; in fact the following conjunctive grammar recognizes the language $\mathcal{L}_{\mathsf{ext}}$;

$$\begin{aligned}
\mathsf{S} \to f[\mathsf{S}'] &\;:=\; s = \mathsf{S}'.s_2 \\
\mathsf{S}' \to f'[\mathsf{R}, \mathsf{L}] &\;:=\; s_1 = \mathsf{R}.s \,\&\, \mathsf{L}.s_1,\ s_2 = \mathsf{L}.s_2 \\
\mathsf{R} \to r[\mathsf{R}] &\;:=\; s = \mathsf{R}.s\ \mathsf{R}.s \\
\mathsf{R} \to a[] &\;:=\; s = \text{`}a\text{'} \\
\mathsf{L} \to l_a[L] &\;:=\; s_1 = \text{`}a\text{'}\ \mathsf{L}.s_1,\ s_2 = \text{`}a\text{'}\ \mathsf{L}.s_2 \\
\mathsf{L} \to l_b[L] &\;:=\; s_1 = \text{`}a\text{'}\ \mathsf{L}.s_1,\ s_2 = \text{`}b\text{'}\ \mathsf{L}.s_2 \\
\mathsf{L} \to e[] &\;:=\; s_1 = \epsilon,\ s_2 = \epsilon
\end{aligned}$$

This grammar consists of two subgrammars; where R recognizes the language a^{2^n}. The second subgrammar L can be seen as a relation on string pairs, where

$\phi.s_1 = a^n$ iff $\phi.s_2 = (a \cup b)^n$. The S' category states that the L argument should have length 2^n, meaning that the s_2 component defines the language $(a \cup b)^{2^n}$. Note that this grammar relies on the possibility of erasing the row s_1, without losing its information.

It is an open question whether there are disjunctive grammars that are not recognizable in polynomial time; or in other words, whether any disjunctive grammar can be translated to an equivalent conjunctive grammar.

5.3.2 Parsing of extensional disjunctive PMCFG

Ranges and extensional disjunction

We can use exactly the same definition of ranges as in section 4.1. The general definition of range disjunction is set union, $\rho_1 \mid \rho_2 = \rho_1 \cup \rho_2$. Since disjunction has a range interpretation, we can extend the notion of range-restriction to also include intersection. Thus, the parsing algorithms for context-free GF in section 4.2 still apply.

Extensional disjunctive PMCFG parsing is not polynomial

Unfortunately, the parsing algorithms for context-free GF are *not* polynomial in the length of the input any more. The reason is that ranges are no longer string-equivalent. In fact, a range can be almost any subset of the universal range $\mathcal{R}_w$; and there are $\mathcal{O}(2^{|w|})$ possible ranges. An exponential number of ranges gives exponential space complexity for the algorithm.

Active parsing of extensional disjunctive PMCFG

Although parsing with general ranges is not polynomial, it is still possible to augment the active parsing algorithm from section 4.4 to handle extensional disjunction. In this algorithm we assume that an intersection $r = \alpha_1 \mid \ldots \mid \alpha_n$ is written as a number of rows in the record, $r = \alpha_1, \ldots, r = \alpha_n$. This is a slight abuse of notation, but the justification is that the inference rules from the original algorithm only need minimal changes.

Union

$$\frac{[\, R;\, \Gamma,\, r = \rho_0,\, r = \rho_1 \bullet,\, \phi;\, \vec{\Gamma}\,]}{[\, R;\, \Gamma,\, r = \rho \bullet,\, \phi;\, \vec{\Gamma}\,]} \left\{ \quad \rho = \rho_0 \cup \rho_1 \right. \tag{5.4}$$

If we have found two linearizations for the same label, we take the union of the linearizations.

Skip

$$\frac{[\,R\,;\;\Gamma,\;r=\rho\bullet,\;r'=\alpha,\;\phi\,;\;\vec{\Gamma}\,]}{[\,R\,;\;\Gamma,\;r=\rho\bullet,\;\phi\,;\;\vec{\Gamma}\,]}\;\begin{cases}\phi=(r'=\alpha',\ldots)\\ \textbf{or}\;r=r'\end{cases}\tag{5.5}$$

But there is also the possibility that some linearization is not possible, so we are allowed to skip a row whenever there are other opportunities for the same label.

Complete

$$\frac{[\,R\,;\;\Gamma,\;r=\rho\bullet,\;r'=\alpha,\;\phi\,;\;\vec{\Gamma}\,]}{[\,R\,;\;\Gamma,\;r=\rho,\;r'=\langle\epsilon\rangle\bullet\alpha,\;\phi\,;\;\vec{\Gamma}\,]}\;\left\{\;\Gamma\neq(\ldots,\;r=\rho_1)\right.\tag{5.6}$$

The only difference to the original Complete rule is the added side condition, stating that the rule must not apply when Union applies.

Predict, Scan and Combine remain as the rules 4.10, 4.12 and 4.13.

5.4 Interleave (‖)

cfg is not an ideal formalism for writing grammars for languages with free or multiple word-order. For this reason more expressive formalisms have been introduced, most notably id/lp grammars (Shieber, 1984). The drawback of the id/lp formalism is that parsing is exponential in the size of the grammar (Barton Jr., 1985).[2] Nederhof et al. (2003) propose to recast id/lp grammars with partially ordered multiset context-free grammars (*poms*-cfg), to generate refined bounds on id/lp parsing complexity. The rules in a *poms*-cfg have *poms*-expressions on the right-hand side, which are a syntactic variant of pomsets (Gischer, 1988).

The main idea with poms-expressions is to introduce the *interleave* operator, (‖). This operator has also been called "merge", "shuffle", "weave" and other things in the contexts of process algebra, concurrency theory and formal language theory (see e.g. Hopcroft and Ullman, 1979; Gischer, 1988).

Definition 5.23 (interleave). Interleave is a non-deterministic linearization operation on sequences defined as $\alpha \parallel \beta = \alpha_1\beta_1\cdots\alpha_n\beta_n$ whenever there are (possibly empty) sequences α_i, β_i such that $\alpha=\alpha_1\ldots\alpha_n$ and $\beta=\beta_1\ldots\beta_n$.

The operation can also be defined inductively via the disjunction operation as;

$$\begin{aligned}a\alpha\parallel b\beta &= a(\alpha\parallel b\beta)\;\mid\;b(a\alpha\parallel\beta)\\ \alpha\parallel\epsilon &= \alpha\\ \epsilon\parallel\beta &= \beta\end{aligned}$$

[2]To be exact, the problem is NP-complete.

This means that interleave is not a strict extension of PMCFG linearizations, provided we use intensional disjunction which itself is a non-strict extension. But expanding an interleave can result in an exponential increase of the size of the linearization. As an example, the expression $a_1 \parallel \cdots \parallel a_n$, saying that the n arguments can come in any order, gives rise to $n!$ distinct disjuncts.

Usefulness of interleave

The main usage of interleaving is to define grammars in free word-order languages. Nederhof et al. (2003) show that there is a direct conversion of grammars written in the ID/LP format to linearizations using interleave.

Example 5.24.

The grammar fragment of verb phrases in the free word-order Makua language (Gazdar et al., 1985, page 48) can be written as follows in the id/lp formalism;

$$\begin{array}{lcll}
\mathsf{VP} & \rightarrow & \mathsf{V} & \mathsf{V} \prec \mathsf{S} \\
\mathsf{VP} & \rightarrow & \mathsf{V, NP} & \\
\mathsf{VP} & \rightarrow & \mathsf{V, S} & \\
\mathsf{VP} & \rightarrow & \mathsf{V, NP, NP} & \\
\mathsf{VP} & \rightarrow & \mathsf{V, NP, PP} & \\
\mathsf{VP} & \rightarrow & \mathsf{V, NP, S} &
\end{array}$$

The rules can be written as an interleaved PMCFG in the following way (adapted from Nederhof et al., 2003);[3]

$$\begin{array}{rcl}
\mathsf{VP} \rightarrow f_1[\mathsf{V}] & := & \mathsf{V} \\
\mathsf{VP} \rightarrow f_2[\mathsf{V, NP}] & := & \mathsf{V} \parallel \mathsf{NP} \\
\mathsf{VP} \rightarrow f_3[\mathsf{V, S}] & := & \mathsf{V} \cdot \mathsf{S} \\
\mathsf{VP} \rightarrow f_4[\mathsf{V, NP^1, NP^2}] & := & \mathsf{V} \parallel \mathsf{NP}^1 \parallel \mathsf{NP}^2 \\
\mathsf{VP} \rightarrow f_5[\mathsf{V, NP, PP}] & := & \mathsf{V} \parallel \mathsf{NP} \parallel \mathsf{PP} \\
\mathsf{VP} \rightarrow f_6[\mathsf{V, NP, S}] & := & \mathsf{V} \cdot \mathsf{S} \parallel \mathsf{NP}
\end{array}$$

▲

5.4.1 Active parsing of interleaved PMCFG

Shieber (1984) has given a direct parsing algorithm for context-free ID/LP grammars, which does not improve on the theoretical parse time complexity for parsing the equivalent CFG, but in practice is much more efficient (Barton Jr., 1985). Nederhof et al. (2003) give a direct parsing algorithm for *poms*-CFG,

[3]For clarity we skip the record labels, since all linearizations are single strings.

which is also exponential in the size of the grammar, but the bounds are even more refined that in Shieber's algorithm. They compile the *poms*-expressions to *poms-automata*, which then are used in their Earley-style parsing algorithm.

Here we modify their algorithm to work with PMCFG grammars. For simplicity we do not compile the linearization expression to a *poms*-automaton, but instead use the expression directly in the parsing algorithm.

Interleave

$$\frac{[\,R\,;\,\Gamma,\,r=\rho\bullet(\alpha_1\,\|\,\cdots\,\|\,X\,\alpha_i\,\|\,\cdots\,\|\,\alpha_n)\,\alpha,\,\phi\,;\,\vec{\Gamma}\,]}{[\,R\,;\,\Gamma,\,r=\rho\bullet X\,(\alpha_1\,\|\,\cdots\,\|\,\alpha_i\,\|\,\cdots\,\|\,\alpha_n)\,\alpha,\,\phi\,;\,\vec{\Gamma}\,]} \tag{5.7}$$

The first component of any part of an interleave can be moved to the front. This is non-deterministic as long as more than one α_i is non-empty.

Merge

$$\frac{[\,R\,;\,\Gamma,\,r=\rho\bullet(\epsilon\,\|\,\cdots\,\|\,\epsilon)\,\alpha,\,\phi\,;\,\vec{\Gamma}\,]}{[\,R\,;\,\Gamma,\,r=\rho\bullet\alpha,\,\phi\,;\,\vec{\Gamma}\,]} \tag{5.8}$$

The interleave of empty strings is empty and can be removed.

Predict, Complete, Scan and Combine remain as the rules 4.10–4.13.

Note that the Interleave rule is not formally correct as it is written. Nested interleaves such as $a\,\|\,(b\,\|\,c)d$ are not handled correctly;

$$a\,\|\,(b\,\|\,c)d \quad\neq\quad a(b\,\|\,c)d \;\mid\; (b\,\|\,c)(a\,\|\,d)$$

Handling nested interleaves

To handle nested interleaves correctly, we need to define *derivatives* over *poms*-expressions, where by $\delta_a(\alpha)$ mean the derivative of α over a. This can be done in a similar way as derivatives for regular expressions (Brzozowski, 1964). As an example, the derivatives of the given expression are

$$\begin{aligned}\delta_a(a\,\|\,(b\,\|\,c)d) &= (b\,\|\,c)d\\ \delta_b(a\,\|\,(b\,\|\,c)d) &= a\,\|\,cd\\ \delta_c(a\,\|\,(b\,\|\,c)d) &= a\,\|\,bd\end{aligned}$$

Note that derivatives can be non-deterministic, as in $\delta_a(ab\,\|\,ac)$ which can be $b\,\|\,ac$ or $ab\,\|\,c$. This constitutes no problem, since the inference rules which use δ_a are non-deterministic themselves.

Now the Interleave rule can be combined into the Scan and Combine rules as follows.

Scan+Interleave

$$\frac{[\, R\, ;\, \Gamma,\, r = \rho \bullet \alpha,\, \phi\, ;\, \vec{\Gamma}\,]}{[\, R\, ;\, \Gamma,\, r = \rho' \bullet \alpha',\, \phi\, ;\, \vec{\Gamma}\,]} \begin{cases} \alpha' = \delta_s(\alpha) \\ \rho' = \rho \cdot \langle s \rangle^w \end{cases} \tag{5.9}$$

Combine+Interleave

$$\frac{[\, R\, ;\, \Gamma,\, r = \rho \bullet \alpha,\, \phi\, ;\, \vec{\Gamma}\,] \quad [\, B\, ;\, \Gamma'\,]}{[\, R\, ;\, \Gamma,\, r = \rho' \bullet \alpha',\, \phi\, ;\, \vec{\Gamma}[i := \Gamma']\,]} \begin{cases} \alpha' = \delta_{B.r'}(\alpha) \\ \rho' = \rho \cdot \Gamma'.r' \\ \Gamma_i \subseteq \Gamma' \end{cases} \tag{5.10}$$

Predict and Combine remain the same as the rules 4.10 and 4.13.

The effect of this algorithm is to simulate a *poms*-automaton via the derivative. As an alternative, the linearizations can also be compiled into automata as is done by Nederhof et al. (2003).

5.5 Summary

In this chapter we gave three different extensions of the pmcfg formalism. The first, intersection, is borrowed from conjunctive grammar (Okhotin, 2001), and is a strict extension since pmcfg is not closed under intersection. We showed that conjunctive pmcfg is equivalent to *simple* literal movement grammar (Groenink, 1997a,b) and range concatenation grammar (Boullier, 2000a,b), meaning that the formalisms exactly characterizes the class of languages recognizable in polynomial time.

The second extension is disjunction, which can have two different interpretations; one intensional and one extensional. The intensional variant is not a strict extension of pmcfg, but the extensional can describe languages which are conjectured not can be described by an ordinary pmcfg. There are some open questions left regarding the extensional disjunction; the first being how the proof sketch can be turned into a correct proof. Another open question is whether disjunctive pmcfg can describe non-polynomial languages.

The third extension is an adaptation of the *poms*-cfg format of Nederhof et al. (2003) to handle pmcfg grammars. The operation is called interleave, and we gave a parsing algorithm for interleaved pmcfg grammars. An application of interleave can be converted to a number of disjunctions, but this reduction can lead to an exponential increase of the grammar size. We instead augmented the active parsing algorithm from chapter 4, with rules to handle interleaves.

Chapter 6

Non-context-free abstract syntax

This final chapter discusses how to handle GF grammars containing higher-order functions or dependent types.

We give an algorithm for converting higher-order functions into first-order functions. The resulting context-free GF grammar is over-generating, since it cannot type-check variable occurrences correctly. We therefore give a procedure for filtering out non-well-formed terms during the conversion from first-order to higher-order parse results.

In the presence of dependent types it is possible to describe undecidable languages (Ranta, 2004a), so the parsing problem is undecidable in general. We nevertheless describe a two-step parsing process for such grammars; first we translate into an overgenerating context-free GF grammar, and parse using that grammar. The resulting parse items are then converted into a logic program, which can be solved by any proof search procedure.

6.1 Higher-order functions

In full GF, arguments to functions can themselves be functions. Functions taking other functions as arguments are called *higher-order* functions, and their corresponding typings are called higher-order categories.

Example 6.1.

The following is a simple grammar for a subset of predicate logic, with quantification over a domain of individuals.

$$\begin{aligned} \mathit{all}, \mathit{some} &: (\mathsf{Ind} \to \mathsf{Prop}) \to \mathsf{Prop} \\ \mathit{equal} &: \mathsf{Ind} \times \mathsf{Ind} \to \mathsf{Prop} \\ a, b, c &: \mathsf{Ind} \end{aligned}$$

And here are some example propositions.

$$\begin{aligned} \mathit{all}(\lambda x.\, \mathit{equal}(x, x)) &: \mathsf{Prop} \\ \mathit{some}(\lambda y.\, \mathit{equal}(b, y) &: \mathsf{Prop} \end{aligned}$$

▲

The presence of higher-order functions gives rise to the question of how to linearize a functional argument $C_1 \times \cdots \times C_n \to B$, a problem that can be solved in different ways. The solution chosen in GF is to pair the linearization of the result category B with *linearizations of variable bindings* representing terms of the argument categories $C_1 \times \cdots \times C_n$. This means that GF automatically infers the linearization type for functional arguments as $\mathsf{Str}^n \times B^\circ$, for a function type $C_1 \times \cdots \times C_n \to B$. For convenience we write elements of the type $(C_1 \times \cdots \times C_n \to B)^\circ$ as $\lambda \langle c_1, \ldots, c_n, b\rangle$.

Example 6.2.

Here is a possible concrete syntax for the given abstract grammar.

$$\begin{aligned} \mathit{all}^\circ(\lambda \langle x, p\rangle) &= \text{‘\textit{for all}’ } x \text{ ‘,’ } p \\ \mathit{some}^\circ(\lambda \langle x, p\rangle) &= \text{‘\textit{there is an}’ } x \text{ ‘\textit{such that}’ } p \\ \mathit{equal}^\circ(x, y) &= x \text{ ‘\textit{is equal to}’ } y \\ a^\circ &= \text{‘\textit{Anna}’} \\ b^\circ &= \text{‘\textit{Bessie}’} \\ c^\circ &= \text{‘\textit{Carlotta}’} \end{aligned}$$

Note that we use $\lambda \langle x, p\rangle$ as syntactic sugar for a pair of the linearization type $(\mathsf{Ind} \to \mathsf{Prop})^\circ = \mathsf{Str} \times \mathsf{Prop}^\circ$.

▲

A formal treatment

To be able to define a concrete syntax for grammars with higher-order functions, we need to introduce some extra notions.

Definition 6.3. The linearization type for a function type $C_1 \times \cdots \times C_n \to B$, is defined as

$$(C_1 \times \cdots \times C_n \to B)^\circ \quad = \quad \mathsf{Str}^n \times B^\circ$$

For each category C the grammar writer needs to define a default linearization for bound variables, which is written

$$\textbf{lindef}\; C(x) \quad = \quad \phi$$

where $\phi : C^\circ$ whenever x is a string. The framework also contains a coercion for variables, where $\hat{x} : \mathsf{Str}$ whenever x is a variable. The most natural way to define this coercion is to view the name of the variable as a string, e.g. $\hat{x} =$ 'x' and $\hat{y}_2 =$ '$y2$'.

Now we can augment the definition of term linearization with two extra cases, for lambda-abstractions and bound variables,

$$\begin{aligned} [\![\lambda x_1 \ldots x_n.\, t]\!] &= \lambda\, \langle \hat{x}_1, \ldots, \hat{x}_n, [\![t]\!] \rangle \\ [\![x]\!] &= \phi \qquad (\textbf{lindef}\; C(\hat{x}) = \phi) \end{aligned}$$

Example 6.4.

The example grammar has to be augmented with default linearizations for each category. But this is trivial; since all linearization types are strings, we can simply declare **lindef** $Ind(x) = x$, and similar for the type $Prop$.

Now we can linearize the example propositions as,

$$\begin{aligned} [\![all(\lambda x.\, equal(x,\, x))]\!] &= all^\circ(\lambda\, \langle \hat{x},\, [\![equal(x,\, x)]\!] \rangle) \\ &= all^\circ(\lambda\, \langle \hat{x},\, equal^\circ(\hat{x},\, \hat{x}) \rangle) \\ &= all^\circ(\lambda\, \langle \text{'}x\text{'},\, equal^\circ(\text{'}x\text{'},\, \text{'}x\text{'}) \rangle) \\ &= \text{'}\textit{for all } x,\ x \textit{ is equal to } x\text{'} \\ [\![some(\lambda y.\, equal(b,\, y))]\!] &= some^\circ(\lambda\, \langle \hat{y},\, [\![equal(b,\, y)]\!] \rangle) \\ &= \text{'}\textit{there is some } y \textit{ such that Bessie is equal to } y\text{'} \end{aligned}$$

▲

6.1.1 Removing higher-order functions from a grammar

In this section we show that a grammar with higher-order functions can be converted to a grammar with a context-free backbone. The resulting grammar is overgenerating in the presence of variables, and we give a procedure for filtering out non-well-formed results.

Algorithm 6.5.

First, add a new category Var for representing bound variables. We assume that there is an infinite supply of terms $v_i : \mathsf{Var}$ ($i \in \mathbb{N}$), representing each possible variable x_i. The linearization type for variables is strings, $\mathsf{Var}^\circ = \mathsf{Str}$, and each term v_i has a linearization definition in terms of its represented variable, $v_i^\circ = \hat{x}_i$.

Then, convert each default linearization **lindef** $C(x) = \phi$ to a function

$$\begin{aligned} \delta_C &: \mathsf{Var} \to C \\ \delta_C^\circ(x) &= \phi \end{aligned}$$

Finally, for each higher-order function such as,

$$h \quad : \quad \vec{B} \times (C_1 \times \cdots \times C_n \to B_k) \times \vec{B}' \to A$$

create a new category $\hat{B}_k = [C_1 \times \cdots \times C_n \to B_k]$ together with a coercion function λ_{B_i},

$$\begin{aligned} \textbf{lincat}\ \hat{B}_k &= \mathsf{Str}^n \times B_k^\circ \\ \lambda_{B_k} &: \mathsf{Var}^n \times B_k \to \hat{B}_k \\ \lambda_{B_k}^\circ(x_1, \ldots, x_n, y) &= \lambda \langle x_1, \ldots, x_n, y \rangle \end{aligned}$$

Replace each occurrence of $C_1 \times \cdots \times C_n \to B_k$ as an argument in a function typing, by the new category $\hat{B}_k$.

▲

The resulting grammar has a context-free backbone since all higher-order functions has been removed. There is an infinite number of terms v_i in Var, but that constitutes no severe problem for parsing, since the number of terms that are used in any given instance is finite, and these terms can be created in a pre-processing phase.

The resulting context-free GF grammar is over-generating, since there is no way of checking that the variables occurring in a term are bound by a preceding lambda-abstraction, and that all occurrences of the same variable have the same type. Therefore we have to check that the variables are bound and type-correct when the terms have been assembled.

Theorem 6.6. *The grammar resulting from algorithm 6.5 recognizes all strings that are recognized by the original grammar.*

Proof. Assume for simplicity that we have a higher-order function of the form,

$$h \quad : \quad \vec{B} \times (C_1 \times \cdots \times C_n \to B_k) \times \vec{B}' \to A$$

The generalization to arbitrary higher-order functions is straightforward. We have to show that there is a corresponding first-order tree for each higher-order tree $h(\vec{t},\, \lambda x_1 \ldots x_n.\, t_k,\, \vec{t'})$, with the same linearization.

The corresponding first-order tree for the higher-order tree $h(\vec{t},\, \lambda x_1 \ldots x_n.\, t_k,\, \vec{t'})$ is,

$$h(\vec{t},\, \lambda_{B_k}(v_1,\, \ldots,\, v_n,\, t_k[x_1/\delta_{C_1}(v_1),\, \ldots,\, x_n/\delta_{C_n}(v_n)]),\, \vec{t'})$$

where $v_1, \ldots, v_n$ are the terms in Var representing the variables $x_1, \ldots, x_n$. Now, the linearization definition of δ_{C_i} says that,

$$[\![\delta_{C_i}(v_i)]\!] = \delta^{\circ}_{C_i}(v^{\circ}_i) = \delta^{\circ}_{C_i}(\hat{x}_i) = \phi_i = [\![x_i]\!]$$

where **lindef** $C_i(\hat{x}_i) = \phi_i$. This implies that the substitution $t_k[x_i/\delta_{C_i}(v_i)]$ does not have effect for any x_i on the linearization; i.e.

$$[\![t_k[x_1/\delta_{C_1}(v_1),\, \ldots,\, x_n/\delta_{C_n}(v_n)]]\!] \quad = \quad [\![t_k]\!]$$

But this in turn gives us,

$$\begin{aligned}
&[\![\lambda_{B_k}(v_1,\, \ldots,\, v_n,\, t_k[x_1/\delta_{C_1}(v_1),\, \ldots,\, x_n/\delta_{C_n}(v_n)])]\!] \\
&\quad = \quad \lambda^{\circ}_{B_k}([\![v_1]\!],\, \ldots,\, [\![v_n]\!],\, [\![t_k[x_1/\delta_{C_1}(v_1),\, \ldots,\, x_n/\delta_{C_n}(v_n)]]\!]) \\
&\quad = \quad \lambda\, \langle \hat{x}_1,\, \ldots,\, \hat{x}_n,\, [\![t_k]\!] \rangle \\
&\quad = \quad [\![\lambda x_1 \ldots x_n.\, t_k]\!]
\end{aligned}$$

□

Converting first-order parse results to higher-order

There is only one possible way to create terms of the new type,

$$\hat{B}_k \quad = \quad [C_1 \times \cdots \times C_n \to B_k]$$

and that is by application of the coercion function λ_{B_k}. So, each term of type $\hat{B}_k$ is of the form $\lambda_{B_k}(v_1,\, \ldots,\, v_n,\, t)$ where $v_1, \ldots, v_n : \mathsf{Var}$ and $t : B_k$. From this term we recreate the function $\lambda x_1 \ldots x_n.\, t'$, where $x_1, \ldots, x_n$ are the variables represented by $v_1, \ldots, v_n$, by substituting,

$$t' \quad = \quad t[\delta_{C_1}(v_1)/x_1,\, \ldots,\, \delta_{C_n}(v_n)/x_n]$$

The resulting function is equivalent to the original term by the argument above, except when t' contains some spurious $\delta_D(v_i)$ after substitution. If it does, the corresponding variable x_i occurs with type D; either it is unbound or it should have had another type.

So, the resulting context-free GF grammar is over-generating, and the recognized terms must be checked for spurious variables, which constitutes a limited form of type-checking.

Example 6.7.

The example grammar now needs to be augmented with the following rules,

$$\begin{aligned} v_i &: \mathsf{Var} \quad (i \in \mathbb{N}) \\ \delta_I &: \mathsf{Var} \to \mathsf{Ind} \\ \delta_P &: \mathsf{Var} \to \mathsf{Prop} \end{aligned}$$

with the variable linearizations v_0° = 'x', v_1° = 'y', v_2° = 'z', etc. The default coercion δ_P will never be used in the grammar, and can be removed. The default linearizations for Ind and Prop are converted to the identical linearization definitions $\delta_I^\circ(x) = x$ and $\delta_P^\circ(x) = x$.

The quantifiers contain the argument type Ind → Prop, from which we introduce the new category $\widehat{\mathsf{P}} = [\mathsf{Ind} \to \mathsf{Prop}]$, with its lambda-coercion λ_P,

$$\begin{aligned} \lambda_P &: \mathsf{Var} \times \mathsf{Prop} \to \widehat{\mathsf{P}} \\ \lambda_P^\circ(x, y) &= \lambda \langle x, y \rangle \end{aligned}$$

Finally we transform the original rules into first-order format,

$$\begin{aligned} \mathit{all}, \mathit{some} &: \widehat{\mathsf{P}} \to \mathsf{Prop} \\ \mathit{equal} &: \mathsf{Ind} \times \mathsf{Ind} \to \mathsf{Prop} \\ a, b, c &: \mathsf{Ind} \end{aligned}$$

retaining their original linearizations.

Now, we can parse the strings '*for all x, x is equal to x*' and '*there is some y such that Bessie is equal to y*', yielding the parse results,

$$\begin{aligned} \mathit{all}(\lambda_P(v_0, \mathit{equal}(\delta_I(v_0), \delta_I(v_0)))) &: \mathsf{Prop} \\ \mathit{some}(\lambda_P(v_1, \mathit{equal}(b, \delta_I(v_1)))) &: \mathsf{Prop} \end{aligned}$$

which can be back-translated into the original parse trees as described above. Note that sentences like *'x is equal to y'* and *'there is some x such that x'*, also are recognized by the first-order grammar; but back-translation fails since the resulting higher-order terms contains spurious variables ($\delta_I(v_0)$ and $\delta_I(v_1)$ for the first sentence, and $\delta_P(v_0)$ for the second).

▲

6.1.2 Higher-order functions as arguments

Suppose we have a higher-order function as argument to a function. An example is the function,

$$h \ : \ D \times (C \times (D \to C) \to B) \to A$$

where the second argument is a higher-order function of type $C \times (D \to C) \to B$.

The first question that arises is how to linearize such a higher-order argument. In GF this is done by assuming that it is a variable like other variables, meaning that the linearization of h has the form,

$$h^\circ(x, \lambda \langle y, f, z \rangle) \quad = \quad \phi$$

where $x : D^\circ$, $z : B^\circ$ and $y, f : \mathsf{Str}$. This treatment means that algorithm 6.5 still works without modifications.

Unfortunately, it is not clear how to handle default linearizations for function types; e.g. what is the default linearization for the function type $D \to C$ above? This is not obvious since functional variables can be applied to arguments; in the example we could apply f to a term of type D, e.g. $f\,x$ is a term of type C.

In the current implementation of GF, there is an ad hoc solution where the application $f\,x$ is linearized as the string '$f\ x$'; but there could perhaps be other alternative choices. These issues have not been fully investigated, since it seems implausible that there will be any linguistic need for more than second-order functions.

6.2 Dependent types

If we have a GF grammar with dependent categories, there is a straightforward two-step parsing process for that grammar. First we simply remove all dependencies from the abstract syntax, thereby getting a grammar with a context-free backbone. This grammar is over-generating, so parsing returns all parse trees we want, but perhaps also some unwanted trees. The second step consists of filtering the parse trees through the original grammar.

The naive way of performing the second step is to extract each parse tree and then check that it is type-correct. However, this can result in extracting a very large number of trees, which are all rejected by the type-checker. In some cases there can even be an infinite number of parse trees, of which only a finite number is correct. Then the filtering algorithm does not even terminate.

Example 6.8.

The following is an example grammar for a fragment of arithmetic with overloaded operators. There are two possible number domains; natural numbers and reals (Nat, $Real$: Dom). Some operations ($plus$) work on any domain, while others ($sqrt$) only work on reals.

$$\begin{array}{rcl}
\mathit{Nat},\ \mathit{Real} & : & \mathsf{Dom} \\
\mathit{plus} & : & (d : \mathsf{Dom}) \times \mathsf{Num}(d) \times \mathsf{Num}(d) \rightarrow \mathsf{Num}(d) \\
\mathit{sqrt} & : & \mathsf{Num}(\mathit{Real}) \rightarrow \mathsf{Num}(\mathit{Real}) \\
c & : & \mathsf{Num}(\mathit{Nat}) \rightarrow \mathsf{Num}(\mathit{Real}) \\
\mathit{one},\ \mathit{two} & : & \mathsf{Num}(\mathit{Nat})
\end{array}$$

$$\begin{array}{rcl}
\mathit{plus}^{\circ}(d,\, x,\, y) & = & x \text{ `}\mathit{plus}\text{' } y \\
\mathit{sqrt}^{\circ}(x) & = & \text{`}\mathit{the\ square\ root\ of}\text{' } x \\
c^{\circ}(x) & = & x \\
\mathit{one}^{\circ} & = & \text{`}\mathit{one}\text{'} \\
\mathit{two}^{\circ} & = & \text{`}\mathit{two}\text{'}
\end{array}$$

Since the abstract type theory does not have overloading, we need a coercion function c from integers to reals. There are three possible terms that linearize to the string $w =$ '*the square root of one plus two*';

$$\begin{array}{ll}
1 & \mathit{sqrt}(\mathit{plus}(\mathit{Real},\ c(\mathit{one}),\ c(\mathit{two}))) \\
2 & \mathit{sqrt}(c(\mathit{plus}(\mathit{Nat},\ \mathit{one},\ \mathit{two}))) \\
3 & \mathit{plus}(\mathit{Real},\ \mathit{sqrt}(c(\mathit{one})),\ c(\mathit{two}))
\end{array}$$

The difference between (1) and (2) is that the first uses integer addition, while the second uses real addition. Both utilize the mathematical term $\sqrt{1+2}$, while (3) is a representation of $\sqrt{1}+2$.

To be able to use the parsing algorithms in chapter 4, we have to remove the dependencies to get the context-free backbone, which looks like this in PMCFG format;

$$\begin{array}{rcl}
\mathsf{Num} \rightarrow \mathit{plus}[\mathsf{Dom},\ \mathsf{Num}^1,\ \mathsf{Num}^2] & := & \mathsf{Num}^1 \text{ `}\mathit{plus}\text{' } \mathsf{Num}^2 \\
\mathsf{Num} \rightarrow \mathit{sqrt}[\mathsf{Num}] & := & \text{`}\mathit{the\ square\ root\ of}\text{' } \mathsf{Num} \\
\mathsf{Num} \rightarrow c[\mathsf{Num}] & := & \mathsf{Num} \\
\mathsf{Num} \rightarrow \mathit{one}[] & := & \text{`}\mathit{one}\text{'} \\
\mathsf{Num} \rightarrow \mathit{two}[] & := & \text{`}\mathit{two}\text{'}
\end{array}$$

Note that the rule for the coercion c is a cyclic rule. Now, parsing the input string w using this grammar results in the following PMCFG chart;

1	$[\,sqrt\,;\, 0\ldots 7\,;\, 4\ldots 7\,]$
2	$[\,sqrt\,;\, 0\ldots 5\,;\, 4\ldots 5\,]$
3	$[\,plus\,;\, 4\ldots 7\,;\, \textbf{?},\, 4\ldots 5,\, 6\ldots 7\,]$
4	$[\,plus\,;\, 0\ldots 7\,;\, \textbf{?},\, 0\ldots 5,\, 6\ldots 7\,]$
5	$[\,one\,;\, 4\ldots 5\,;\ ]$
6	$[\,two\,;\, 6\ldots 7\,;\ ]$
7	$[\,c\,;\, 0\ldots 7\,;\, 0\ldots 7\,]$
8	$[\,c\,;\, 0\ldots 5\,;\, 0\ldots 5\,]$
9	$[\,c\,;\, 4\ldots 7\,;\, 4\ldots 7\,]$
10	$[\,c\,;\, 4\ldots 5\,;\, 4\ldots 5\,]$
11	$[\,c\,;\, 6\ldots 7\,;\, 6\ldots 7\,]$

Note that items (3) and (4) have metavariables for the linearization of the domain, since the rule for *plus* is suppressing. The items (7)–(11) are all cyclic items which can be applied on any part of a parse tree, any number of times. So, there are infinitely many parse trees that can be constructed from this chart, of the following two forms;

$$c^*(sqrt(c^*(plus(c^*(one),\, c^*(two)))))$$
$$c^*(plus(c^*(sqrt(c^*(one)),\, c^*(two))))$$

Of these infinitely many trees, only three are correct according to the original grammar.

▲

6.2.1 Type checking as proof search

In this section we describe how to do the second step on the resulting parse chart, instead of each parse tree. The idea is to convert the chart into Horn clauses, which can be solved by any proof search procedure, e.g. standard PROLOG.

Converting to Horn clauses

Algorithm 6.9.

Convert each chart item,

$$[\,A \to f[A_1,\, \ldots,\, A_\delta]\,;\, \phi\,;\, \phi_1,\, \ldots,\, \phi_\delta\,]$$

where f has the abstract typing,

$$\begin{aligned} f \quad : \quad & (x_1 : A_1) \times (x_2 : A_2[x_1]) \times \cdots \times (x_\delta : A_\delta[x_1,\, \ldots,\, x_{\delta-1}]) \\ & \to A[x_1,\, \ldots,\, x_\delta] \end{aligned}$$

into the following Horn clause (where $t : A \Rightarrow \phi$ is just syntactic sugar for a 3-tuple);

$$\begin{aligned} f(x_1, \ldots, x_\delta) : A[x_1, \ldots, x_\delta] \Rightarrow \phi \quad \vdash \quad & x_1 : A_1 \Rightarrow \phi_1, \\ & x_2 : A_2[x_1] \Rightarrow \phi_2, \\ & \ldots, \\ & x_\delta : A_\delta[x_1, \ldots, x_{\delta-1}] \Rightarrow \phi_\delta \end{aligned}$$

Metavariables in a linearization ϕ_i are treated as anonymous logical variables. ▲

This algorithm works fine for non-suppressing grammars; but if the rule for f above is suppressing, then some linearization ϕ_i in the resulting clause might be completely uninstantiated, $\phi_i = \mathbf{?}$. This means that the term represented by x_i is suppressed, and the corresponding predicate call can (and must) be removed from the clause.

Algorithm 6.10.

From all clauses,

$$\Phi \quad \vdash \quad \Phi_1, \ldots, \Phi_\delta$$

remove each predicate call Φ_i with an uninstantiated linearization,

$$\Phi_i \quad = \quad x_i : A_i[\ldots] \Rightarrow \mathbf{?}$$

▲

Querying the logic program

Now, given the input string w and the starting category S, we can try to prove the query,[1]

$$\vdash \quad x : S \Rightarrow \langle w \rangle$$

where x is an unbound logical variable. If the query is true, the string is accepted by the grammar, and each possible instantiation of x is a correct parse tree.

While the first step, parsing the underlying PMCFG grammar, always terminates in polynomial time; it is not sure that the second step will. It might take exponential time to extract a solution from the resulting logic program, or it might even be non-terminating. Also, the time taken and the termination may depend on the proof strategy of the theorem solver.

[1]Note that the query uses the range interpretation of section 4.1, which presupposes that the chart items also use range linearizations. If the chart uses string linearizations, we only have to replace $\langle w \rangle$ by w in the query.

Example 6.11.

The chart from the previous example, is translated by the algorithm into the following logic program;

$$\begin{array}{rrcl}
1 & \mathit{sqrt}(x) : \mathsf{Num}(\mathit{Real}) \Rightarrow 0\ldots7 & \vdash & x : \mathsf{Num}(\mathit{Real}) \Rightarrow 4\ldots7 \\
2 & \mathit{sqrt}(x) : \mathsf{Num}(\mathit{Real}) \Rightarrow 0\ldots5 & \vdash & x : \mathsf{Num}(\mathit{Real}) \Rightarrow 4\ldots5 \\
3 & \mathit{plus}(d,\, x,\, y) : \mathsf{Num}(d) \Rightarrow 4\ldots7 & \vdash & x : \mathsf{Num}(d) \Rightarrow 4\ldots5, \\
 & & & y : \mathsf{Num}(d) \Rightarrow 6\ldots7 \\
4 & \mathit{plus}(d,\, x,\, y) : \mathsf{Num}(d) \Rightarrow 0\ldots7 & \vdash & x : \mathsf{Num}(d) \Rightarrow 0\ldots5, \\
 & & & y : \mathsf{Num}(d) \Rightarrow 6\ldots7 \\
5 & \mathit{one} : \mathsf{Num}(\mathit{Nat}) \Rightarrow 4\ldots5 & \vdash & \epsilon \\
6 & \mathit{two} : \mathsf{Num}(\mathit{Nat}) \Rightarrow 6\ldots7 & \vdash & \epsilon \\
7 & c(x) : \mathsf{Num}(\mathit{Real}) \Rightarrow 0\ldots7 & \vdash & x : \mathsf{Num}(\mathit{Nat}) \Rightarrow 0\ldots7 \\
8 & c(x) : \mathsf{Num}(\mathit{Real}) \Rightarrow 0\ldots5 & \vdash & x : \mathsf{Num}(\mathit{Nat}) \Rightarrow 0\ldots5 \\
9 & c(x) : \mathsf{Num}(\mathit{Real}) \Rightarrow 4\ldots7 & \vdash & x : \mathsf{Num}(\mathit{Nat}) \Rightarrow 4\ldots7 \\
10 & c(x) : \mathsf{Num}(\mathit{Real}) \Rightarrow 4\ldots5 & \vdash & x : \mathsf{Num}(\mathit{Nat}) \Rightarrow 4\ldots5 \\
11 & c(x) : \mathsf{Num}(\mathit{Real}) \Rightarrow 6\ldots7 & \vdash & x : \mathsf{Num}(\mathit{Nat}) \Rightarrow 6\ldots7
\end{array}$$

The goal to prove is,

$$\vdash \quad x : Num(d) \Rightarrow 0\ldots7$$

and using a simple theorem prover such as PROLOG, we get the following answers;

$$\begin{array}{ll}
1 & d = \mathit{Real},\ x = \mathit{sqrt}(\mathit{plus}(\mathit{Real},\ c(\mathit{one}),\ c(\mathit{two}))) \\
2 & d = \mathit{Real},\ x = \mathit{sqrt}(c(\mathit{plus}(\mathit{Nat},\ \mathit{one},\ \mathit{two}))) \\
3 & d = \mathit{Real},\ x = \mathit{plus}(\mathit{Real},\ \mathit{sqrt}(c(\mathit{one})),\ c(\mathit{two}))
\end{array}$$

▲

Horn clauses as a representation of a chart

The logic program resulting from the algorithm in this section can be seen as a compact representation of the set of parse trees. But then there is an analogy to Lang (1994); where *parsing* in the sense "constructing a compact representation of the parse trees" takes polynomial time, while *recognition* in the sense "testing whether the input string is recognized by the grammar" is undecidable in the worst case. Another example where parsing is harder than recognition is INDEXED GRAMMAR (Aho, 1968), which was noted by Lang (1994).

Dependencies ranging over finite types

A special case of dependent types which is not considered in this chapter, is when the dependencies range over finite types. The type $\mathsf{Num}(d)$ in example 6.8 is an example of this, since the variable $d : \mathsf{Dom}$ has a finite range, *Nat*, *Real*. In this case we can treat the types $\mathsf{Num}(Nat)$ and $\mathsf{Num}(Real)$ as basic categories, and not dependent types. The function *plus* will have to be split into two functions; one for integers and one for reals. As a final remark, we note that treatment of finite dependencies is similar to the translation of context-free GF to PMCFG in chapter 3, where finite parameters are moved into the categories.

6.2.2 Dependent types and higher-order functions

A GF grammar can contain functions which are both higher-order and have dependent typings. These grammars can be parsed using the techniques in this chapter. We only have to know how GF handles default linearizations for dependent types.

For the same reason why each instance of a dependent type has the same linearization type, the default linearization is the same for each instance of a dependent type. This means that the default linearization for a dependent type $C(\vec{x})\ [\vec{x} : \vec{D}]$ can be specified as,

$$\textbf{lindef}\ C(\_)\ (y) \quad = \quad \phi$$

where $\phi : C(\_)^{\circ}$ whenever y is a string.

Example 6.12.

If we want to add propositions to the grammar for simple arithmetic, we have to introduce a higher-order function with a dependent type for the quantifiers;

$$\begin{array}{rcl} \mathit{all},\ \mathit{some} & : & (d : \mathsf{Dom}) \times (\mathsf{Num}(d) \to \mathsf{Prop}) \to \mathsf{Prop} \\ \mathit{equal} & : & (d : \mathsf{Dom}) \times \mathsf{Num}(d) \times \mathsf{Num}(d) \to \mathsf{Prop} \\ \mathit{all}^{\circ}(d,\ \lambda\,\langle x,\ p\rangle) & = & \text{‘}\mathit{for\ all}\text{’}\ x\ \text{‘},\text{’}\ p \\ \mathit{some}^{\circ}(d,\ \lambda\,\langle x,\ p\rangle) & = & \text{‘}\mathit{there\ is\ an}\text{’}\ x\ \text{‘}\mathit{such\ that}\text{’}\ p \\ \mathit{equal}^{\circ}(d,\ x,\ y) & = & x\ \text{‘}\mathit{is\ equal\ to}\text{’}\ y \end{array}$$

Using these functions, we can form propositions like,

$$p \;=\; \mathit{some}(\mathit{Nat},\ \lambda x.\ \mathit{equal}(\mathit{Real},\ c(x),\ \mathit{sqrt}(c(x)))) \quad : \quad \mathsf{Prop}$$

saying that $x = \sqrt{x}$ for some integer x;

$$[\![p]\!] \quad = \quad \text{‘}\mathit{there\ is\ an\ x\ such\ that\ x\ is\ equal\ to\ the\ square\ root\ of\ x}\text{’}$$

Note that since the linearization functions drop the domain information, the string does not say that the number is an integer.

The default linearizations for numbers is the trivial identity function as before;

$$\textbf{lindef}\ \mathsf{Num}(\_)\ (y) \quad = \quad y$$

▲

The idea is to first translate higher-order functions to first-order; then parse the context-free backbone, and convert the resulting chart into a logic program.

To translate a higher-order function with a dependent functional argument, e.g.

$$h \quad : \quad (x,\, y : D) \times (C_1[x] \times C_2[x,\, y] \rightarrow B[y]) \rightarrow A$$

we move the variables x, y outside the new category $\hat{B}$,

$$\hat{B}(x,\, y) \quad = \quad [C_1 \times C_2 \rightarrow B](x,\, y)$$

together with the coercion function λ_B;

$$\begin{aligned} \textbf{lincat}\ \hat{B} \quad &= \quad \mathsf{Str}^2 \times B^\circ \\ \lambda_B \quad &: \quad (x,\, y : D) \times \mathsf{Var} \times \mathsf{Var} \times B[y] \rightarrow \hat{B}(x,\, y) \\ \lambda_B(x,\, y,\, c_1,\, c_2,\, b) \quad &= \quad \lambda\, \langle x,\, y,\, c_1,\, c_2,\, b\rangle \end{aligned}$$

Furthermore, the default linearizations **lindef** $C_i(\_)\ (x) = \phi_i$ are translated to functions,

$$\begin{aligned} \delta_{C_1} \quad &: \quad (x : D) \times \mathsf{Var} \rightarrow C_1[x] \\ \delta_{C_2} \quad &: \quad (x,\, y : D) \times \mathsf{Var} \rightarrow C_2[x,\, y] \\ \delta^\circ_{C_1}(\_,\, x) \quad &= \quad \phi_1 \\ \delta^\circ_{C_2}(\_,\, \_,\, x) \quad &= \quad \phi_2 \end{aligned}$$

With these modifications, algorithm 6.5 can be applied to dependent higher-order functions as well.

Example 6.13.

The arithmetic grammar with propositions is translated to first-order form by introducing the category $\widehat{\mathsf{P}}(d)$ $[d : \mathsf{Dom}]$, where $\widehat{\mathsf{P}} = [\mathsf{Num} \rightarrow \mathsf{Prop}]$, and the following rules;

$$\begin{aligned} v_i \quad &: \quad \mathsf{Var} \qquad (i \in \mathbb{N}) \\ \delta_N \quad &: \quad (d : \mathsf{Dom}) \times \mathsf{Var} \rightarrow \mathsf{Num}(d) \\ \delta_N(\_,\, x) \quad &= \quad x \\ \lambda_P \quad &: \quad (d : \mathsf{Dom}) \times \mathsf{Var} \times \mathsf{Prop} \rightarrow \widehat{\mathsf{P}}(d) \\ \lambda_P(d,\, x,\, p) \quad &= \quad \lambda\, \langle d,\, x,\, p\rangle \end{aligned}$$

Now we can transform the higher-order quantifiers into first-order format,

$$\mathit{all},\ \mathit{some} \quad : \quad (d : \mathsf{Dom}) \times \widehat{\mathsf{P}}(d) \rightarrow \mathsf{Prop}$$

keeping their original linearizations.

Parsing the string $[\![p]\!]$ above yields two parse results since the string does not contain information whether the variable is an integer or a real number;

1 $some(Nat,\ \lambda_P(Nat,\ v_0,\ equal(Real,\ c(\delta_N(v_0)),\ sqrt(c(\delta_N(v_0))))))$

2 $some(Real,\ \lambda_P(Real,\ v_0,\ equal(Real,\ \delta_N(v_0),\ sqrt(\delta_N(v_0)))))$

Back-translating the terms to higher-order form as in the proof of theorem 6.6, results in the original term p and a similar term in which x is a real number variable. ▲

6.3 Limitations of the approach in this chapter

The approach to GF parsing taken in this chapter is thus to;

1. First translate higher-order functions to first-order;
2. Then parse using the context-free backbone and convert the resulting chart to a logic program.

What are the limitations of this approach; or in other words, which GF grammars *cannot* be handled by this two-step process?

6.3.1 Function definitions

One feature of the type theory of GF, that cannot be handled by the approach in this chapter, is abstract function definitions. Recall that a GF grammar can contain definitions of the form **def** $f\,\vec{x} = t$. During type-checking, it can sometimes be necessary to reduce terms by such definitions, which is not handled in our approach.

Example 6.14.

Example grammar 6.12 for propositions of arithmetic, could be augmented with definitions of the constants *one*, *two*, and integer addition,

$$\begin{aligned}
\textbf{def}\ one &= succ(zero)\\
\textbf{def}\ two &= succ(one)\\
\textbf{def}\ plus(Nat,\ zero,\ y) &= y\\
\textbf{def}\ plus(Nat,\ succ(x),\ y) &= succ(plus(x,\ y))
\end{aligned}$$

where the typings for the natural number constructors *zero* and *succ* are standard;

$$\begin{aligned}
zero &: \mathsf{Num}(Nat)\\
succ &: \mathsf{Num}(Nat) \to \mathsf{Num}(Nat)
\end{aligned}$$

Assume also that we introduce the dependent type of proofs of propositions,

$$\mathsf{Proof}(p)\ \mathsf{Type}\ [p : \mathsf{Prop}]$$

with the following instances for integer equality;

$$\begin{aligned} eq_0 &: \mathsf{Proof}(\mathit{equal}(\mathit{Nat},\, \mathit{zero},\, \mathit{zero})) \\ eq_s &: (m, n : \mathsf{Num}(\mathit{Nat})) \times \mathsf{Proof}(\mathit{equal}(\mathit{Nat},\, m,\, n)) \\ &\qquad \rightarrow \mathsf{Proof}(\mathit{equal}(\mathit{Nat},\, \mathit{succ}(m),\, \mathit{succ}(n))) \end{aligned}$$

Now, the proposition $2 = 1 + 1$,

$$P \;=\; \mathit{equal}(\mathit{Nat},\, \mathit{two},\, \mathit{plus}(\mathit{one},\, \mathit{one})) \quad : \quad \mathsf{Prop}$$

has a proof which can be written;

$$p \;=\; eq_s(\mathit{one},\, \mathit{one},\, eq_s(\mathit{zero},\, \mathit{zero},\, eq_0)) \quad : \quad \mathsf{Proof}(P)$$

However, to check that the proof is correct, i.e. to type-check the proof, it is necessary to expand the function definitions for *one* and *two* in p and P.

▲

Assuming that we have linearization definitions for all proof terms and propositions, we could in principle parse a linearization of a proof, and succeed if the proof is correct. But, then we need to expand function definitions during type-checking, and cannot use the approach of this chapter. The only current option is then to resort to extracting all possible parse trees and type-checking them one at the time.

6.3.2 Lambda-abstractions in typings

Another feature of the type theory, is the possibility to have lambda-abstractions, representing anonymous functions, in function typings. When type-checking terms of this kind, it might be necessary to postpone calculation of applications of anonymous functions until the type gets instantiated, during type checking. As with function definitions, this feature is also handled in the current GF implementation by extracting all parse trees and type-checking the one at the time.

Example 6.15.

Useful linguistic examples of functions having anonymous functions in the typing are difficult to find. Even in logic they are sparse. One example is the third-order elimination rule of universal quantification (Π-elimination, also known as *funsplit*). We do not give the typing of that function here, but refer to Nordström et al. (1990, p. 56).

▲

6.4 Summary

This final chapter discussed how to handle GF grammars containing higher-order functions or dependent types.

We gave an algorithm for converting higher-order functions into first-order functions. The resulting context-free GF grammar is over-generating, since it cannot type-check variable occurrences correctly. We therefore gave a procedure for filtering out non-well-formed terms during the conversion from first-order to higher-order parse results. Our current solution is not entirely satisfactory, since it amounts to generating every possible first-order parse tree, and type-check the term during the conversion to a higher-order term. A better solution would be to do the type-checking implicitly during the parsing process, or alternatively during the extraction of first-order terms from the chart.

In the presence of dependent types it is possible to describe undecidable languages (Ranta, 2004a), so the parsing problem is undecidable in general. We nevertheless described a two-step parsing process for such grammars; first translate into an overgenerating context-free GF grammar, by stripping off dependencies, and parse using that grammar. This grammar can be parsed using the algorithms in chapter 4, and then the resulting parse items are converted into a logic program consisting of Horn clauses. The logic program can finally be solved by a first-order theorem prover.

There is no guarantee that a grammar with dependent types always can be parsed, but formulating the solutions as a logic program often reduces the search space, compared to the alternative of generating all possible terms and type-checking them one at the time.

Bibliography

ACL (2004). *Incremental Parsing: Bringing Engineering and Cognition Together*, ACL 2004 Workshop, Barcelona, Spain. `http://www.acl2004.org/`

Aho, A. (1968). Indexed grammars—an extension to context-free grammars. *Journal of the ACM*, 15:647–671.

Aho, A., Sethi, R., and Ullman, J. (1986). *Compilers – Principles, Techniques and Tools*. Addison-Wesley.

Ajdukiewicz, K. (1935). Die syntaktische Konnexität. *Studia Philosophica*, 1:1–27.

Bar-Hillel, Y. (1953). A quasi-arithmetical notation for syntactic description. *Language*, 29:47–58.

Bar-Hillel, Y., Perles, M., and Shamir, E. (1964). On formal properties of simple phrase structure grammars. In Bar-Hillel, Y., editor, *Language and Information: Selected Essays on their Theory and Application*, chapter 9, pages 116–150. Addison-Wesley.

Barton Jr., G. E. (1985). The computational difficulty of ID/LP parsing. In *23rd Meeting of the Association for Computational Linguistics*, pages 76–81, Chicago, Illinois.

Becker, T. (1994). *HyTAG: A New Type of Tree Adjoining Grammars*. PhD thesis, Universität des Saarlandes.

Bertsch, E. and Nederhof, M.-J. (2001). On the complexity of some extensions of RCG parsing. In *7th International Workshop on Parsing Technologies*, pages 66–77.

Billot, S. and Lang, B. (1989). The structure of shared forests in ambiguous parsing. In *27th Meeting of the Association for Computational Linguistics*, pages 143–151, Vancouver, Canada.

Boullier, P. (2000a). A cubic-time extension of context-free grammars. *Grammars*, 3:111–131.

Boullier, P. (2000b). Range concatenation grammars. In *6th International Workshop on Parsing Technologies*, pages 53–64, Trento, Italy.

Bresnan, J. and Kaplan, R. (1982). Lexical-functional grammar: A formal system for grammatical representation. In Bresnan, J., editor, *The Mental Representation of Grammatical Relations*, pages 173–281. MIT Press, Cambridge, MA.

Brzozowski, J. A. (1964). Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494.

Carroll, J. (2003). Parsing. In Mitkov, R., editor, *The Oxford Handbook of Computational Linguistics*, chapter 12, pages 233–248. Oxford University Press.

Chiang, D. (2001). Constraints on strong generative power. In *39th Meeting of the Association for Computational Linguistics*, pages 124–131.

Chiang, D. (2004). *Evaluating Grammar Formalisms for Applications to Natural Language Processing and Biological Sequence Analysis.* PhD thesis, University of Pennsylvania.

Chomsky, N. (1957). *Syntactic Structures.* Mouton, The Hague.

Chomsky, N. (1959). On certain formal properties of grammars. *Information and Control*, 2:137–167.

Chomsky, N. (1965). *Aspects of the Theory of Syntax.* MIT Press, Cambridge, MA.

Coppersmith, D. and Winograd, S. (1990). Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280.

Coq (1999). *The Coq Proof Assistant Reference Manual.* The Coq Development Team. Available at `http://pauillac.inria.fr/coq/`

Curry, H. B. (1963). Some logical aspects of grammatical structure. In Jacobson, R., editor, *Structure of Language and its Mathematical Aspects: Proceedings of the 12th Symposium in Applied Mathematics*, pages 56–68. American Mathematical Society.

Daniels, M. and Meurers, D. (2002). Improving the efficiency of parsing with discontinuous constituents. In *NLULP-02: 7th International Workshop on Natural Language Understanding and Logic Programming*, Copenhagen, Denmark.

Debusmann, R., Duchier, D., and Kruijff, G.-J. M. (2004). Extensible dependency grammar: A new methodology. In *COLING 2004 Workshop on Recent Advances in Dependency Grammar*.

Donzeau-Gouge, V., Huet, G., Kahn, G., Lang, B., and Levy, J. J. (1975). A structure-oriented program editor: a first step towards computer assisted programming. In *International Computing Symposium (ICS'75)*.

Dymetman, M., Lux, V., and Ranta, A. (2000). XML and multilingual document authoring: Convergent trends. In *COLING*, pages 243–249, Saarbrücken, Germany.

Earley, J. (1970). An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102.

Gaifman, H. (1965). Dependency systems and phrase-structure systems. *Information and Control*, 8:304–337.

Gazdar, G. (1987). Applicability of indexed grammars to natural languages. In Reyle, U. and Rohrer, C., editors, *Natural Language Parsing and Linguistic Theories*, pages 69–94. D. Reidel Publishing Company.

Gazdar, G., Klein, E., Pullum, G., and Sag, I. (1985). *Generalized Phrase Structure Grammar*. Basil Blackwell, Oxford, England.

GF (2004). The Grammatical Framework homepage. Located at `http://www.cs.chalmers.se/~aarne/GF`

Ginsburg, S. (1975). *Algebraic and Automata-Theoretic Properties of Formal Languages*. North-Holland/Elsevier.

Gischer, J. L. (1988). The equational theory of pomsets. *Theoretical Computer Science*, 61(2–3):199–224.

Graham, S., Harrison, M., and Ruzzo, W. (1980). An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems*, 2(3):415–462.

Groenink, A. (1997a). Mild context-sensitivity and tuple-based generalizations of context-free grammar. *Linguistics and Philosophy*, 20:607–636.

Groenink, A. (1997b). *Surface without Structure — Word order and tractability issues in natural language analysis*. PhD thesis, Utrecht University.

Hallgren, T. and Ranta, A. (2000). An extensible proof text editor. In Parigot, M. and Voronkov, A., editors, *LPAR-2000*, volume 1955 of *LNCS/LNAI*, pages 70–84. Springer.

Harper, R., Honsell, F., and Plotkin, G. (1993). A framework for defining logics. *Journal of the ACM*, 40(1):143–184.

Hays, D. (1964). Dependency theory: A formalism and some observations. *Language*, 40:511–525.

Hopcroft, J. and Ullman, J. (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

Hudson, R. (1990). *English Word Grammar*. Blackwell.

Hähnle, R., Johannisson, K., and Ranta, A. (2002). An authoring tool for informal and formal requirements specifications. In Kutsche, R.-D. and Weber, H., editors, *Fundamental Approaches to Software Engineering*, volume 2306 of *LNCS*, pages 233–248. Springer.

Joshi, A. (1985). How much context-sensitivity is necessary for characterizing structural descriptions — tree adjoining grammars. In Dowty, D., Karttunen, L., and Zwicky, A., editors, *Natural Language Processing: Psycholinguistic, Computational and Theoretical Perspectives*, pages 206–250. Cambridge University Press, New York.

Joshi, A. and Schabes, Y. (1997). Tree-adjoining grammars. In Rozenberg, G. and Salomaa, A., editors, *Handbook of Formal Languages. Vol 3: Beyond Words*, chapter 2, pages 69–123. Springer-Verlag, Berlin/Heidelberg/New York.

Joshi, A. K., Levy, L. S., and Takahashi, M. (1975). Tree adjunct grammars. *Journal of Computer and System Sciences*, 10(1):136–163.

Karttunen, L., Chanod, J.-P., Grefenstette, G., and Schiller, A. (1996). Regular expressions for language engineering. *Natural Language Engineering*, 2(4):305–328.

Kasami, T. (1965). An efficient recognition and syntax algorithm for context-free languages. Technical Report AFCLR-65-758, Air Force Cambridge Research Laboratory, Bedford, MA.

Kasami, T., Seki, H., and Fujii, M. (1988). Generalized context-free grammars and multiple context-free grammars. *IEICE Transactions*, J71-D-I(5):758–765.

Kay, M. (1986). Algorithm schemata and data structures in syntactic processing. In Grosz, B., Jones, K., and Webber, B., editors, *Readings in Natural Language Processing*, pages 35–70. Morgan Kaufman Publishers, Los Altos, CA.

Khegai, J., Nordström, B., and Ranta, A. (2003). Multilingual syntax editing in GF. In Gelbukh, A., editor, *CICLing-2003: Intelligent Text Processing and Computational Linguistics*, LNCS 2588, pages 453–464. Springer.

Kilbury, J. (1985). Chart parsing and the Earley algorithm. In Klenk, U., editor, *Kontextfreie Syntaxen und wervandte Systeme*. Niemeyer, Tübingen, Germany.

Knuth, D. E. (1965). On the translation of languages from left to right. *Information and Control*, 8:607–639.

Lager, T. and Kronlid, F. (2004). The Current platform: Building conversational agents in Oz. In *2nd International Mozart/Oz Conference*.

Lambek, J. (1958). The mathematics of sentence structure. *American Mathematical Monthly*, 65:154–170.

Landin, P. J. (1966). The next 700 programming languages. *Communications of the ACM*, 9(3):157–166.

Lang, B. (1974). Deterministic techniques for efficient non-deterministic parsers. In Loeckx, J., editor, *Proceedings of the 2nd Colloquium on Automata, Languages and Programming*, volume 14 of *LNCS*, pages 255–269. Springer-Verlag.

Lang, B. (1991). Towards a uniform framework for parsing. In Tomita, M., editor, *Current Issues in Parsing Technology*, pages 153–171. Kluwer.

Lang, B. (1994). Recognition can be harder than parsing. *Computational Intelligence*, 10(4):486–494.

Lee, L. (2002). Fast context-free grammar parsing requires fast Boolean matrix multiplication. *Journal of the ACM*, 49(1):1–15.

Lundwall, S. J. (1974). *Alice, Alice!* Delta Förlag, Stockholm, Sweden.

Magnusson, L. and Nordström, B. (1994). The ALF proof editor and its proof engine. In *Types for Proofs and Program*, volume 806 of *LNCS*, pages 213–237. Springer.

Martin-Löf, P. (1984). *Intuitionistic Type Theory.* Bibliopolis, Napoli.

McCarthy, J. (1963). Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, Amsterdam. North-Holland.

Mel'cuk, I. (1988). *Dependency Syntax: Theory and Practice.* State University of New York Press.

Miller, P. H. (1999). *Strong Generative Capacity.* Number 103 in CSLI lecture notes. CSLI Publications, Stanford, CA.

Milner, R., Tofte, M., Harper, R., and MacQueen, D. (1997). *The Definition of Standard ML – Revised.* MIT Press, Cambridge, MA.

Mohri, M. (1997). Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–312.

Montague, R. (1974). *Formal Philosophy.* Yale University Press, New Haven. Collected papers edited by R. Thomason.

Morrill, G. (1994). *Type Logical Grammar: Categorial Logic of Signs.* Dordrecht.

Mäenpää, P. and Ranta, A. (1999). The type theory and type checker of GF. In *PLI-1999 workshop on Logical Frameworks and Meta-languages*, Paris, France.

Nakanishi, R., Takada, K., Nii, H., and Seki, H. (1998). Efficient recognition algorithms for parallel multiple context-free languages and multiple context-free languages. *IEICE Transactions*, E81–D(11):1148–1161.

Nakanishi, R., Takada, K., and Seki, H. (1997). An efficient recognition algorithm for multiple context-free languages. In *MOL5: 5th Meeting on the Mathematics of Language*, pages 119–123, Saarbrücken, Germany.

Nederhof, M.-J. and Satta, G. (1996). Efficient tabular LR parsing. In *34th Meeting of the Association for Computational Linguistics*, pages 239–246, Santa Cruz, California.

Nederhof, M.-J. and Satta, G. (2004). Tabular parsing. In Martin-Vide, C., Mitrana, V., and Paun, G., editors, *Formal Languages and Applications*, volume 148 of *Studies in Fuzziness and Soft Computing*, pages 529–549. Springer-Verlag.

Nederhof, M.-J., Satta, G., and Shieber, S. (2003). Partially ordered multiset context-free grammars and free-word-order parsing. In *8th International Workshop on Parsing Technologies*, pages 171–182, Nancy, France.

Nordström, B., Petersson, K., and Smith, J. (1990). *Programming in Martin-Löf's Type Theory.* Oxford University Press.

Okhotin, A. (2001). Conjunctive grammars. *Journal of Automata, Languages and Combinatorics*, 6(4):519–535.

Peyton Jones, S. (2003). *Haskell 98 Language and Libraries.* Cambridge University Press, New York.

Pollard, C. (1984). *Generalised Phrase Structure Grammars, Head Grammars and Natural Language.* PhD thesis, Stanford University.

Pollard, C. and Sag, I. (1994). *Head-Driven Phrase Structure Grammar.* University of Chicago Press.

Rajasekaran, S. and Yooseph, S. (1995). TAL recognition in $\mathcal{O}(m(n^2))$ time. In *33rd Meeting of the Association for Computational Linguistics*, pages 166–173.

Ranta, A. (1994). *Type-Theoretical Grammar.* Oxford University Press.

Ranta, A. (2004a). Grammatical Framework, a type-theoretical grammar formalism. *Journal of Functional Programming*, 14(2):145–189.

Ranta, A. (2004b). Modular grammar engineering in GF. Submitted.

Ranta, A. and Cooper, R. (2004). Dialogue systems as proof editors. *Journal of Logic, Language and Information*, 13(2):225–240.

Rayner, M., Dowding, J., and Hockey, B. A. (2001). A baseline method for compiling typed unification grammars into context free language models. In *EUROSPEECH 2001*, Aalborg, Denmark.

Reape, M. (1991). Parsing bounded discontinuous constituents: Generalisations of some common algorithms. In Reape, M., editor, *Word Order in Germanic and Parsing*, pages 41–70. Centre for Cognitive Science, Edinburgh.

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–49.

Satta, G. (1994). Tree adjoining grammar parsing and boolean matrix multiplication. *Computational Linguistics*, 20(2):173–192.

Seki, H., Matsumara, T., Fujii, M., and Kasami, T. (1991). On multiple context-free grammars. *Theoretical Computer Science*, 88:191–229.

Shieber, S. (1984). Direct parsing of ID/LP grammars. *Linguistics and Philosophy*, 7(2):135–154.

Shieber, S. (1985). Evidence against the context-freeness of natural language. *Computational Linguistics*, 20(2):173–192.

Shieber, S., Schabes, Y., and Pereira, F. (1995). Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1–2):3–36.

Sikkel, K. (1997a). Parsing of context-free languages. In Rozenberg, G. and Salomaa, A., editors, *The Handbook of Formal Languages*, volume II. Springer-Verlag, Berlin.

Sikkel, K. (1997b). *Parsing Schemata.* Springer Verlag.

Sikkel, K. (1998). Parsing schemata and correctness of parsing algorithms. *Theoretical Computer Science*, 199:87–103.

Steedman, M. (1985). Dependency and coordination in the grammar of Dutch and English. *Language*, 61:523–568.

Steedman, M. (1986). Combinators and grammars. In Oehrle, R., Bach, E., and Wheeler, D., editors, *Categorial Grammars and Natural Language Structures*, pages 417–442. Foris, Dordrecht.

Teitelbaum, T. and Reps, T. (1981). The Cornell Program Synthesizer: a syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573.

Tomita, M. (1986). *Efficient Parsing for Natural Language*. Kluwer Academic Press.

Valiant, L. (1975). General context-free recognition in less than cubic time. *Journal of Computer and Systems Sciences*, 10(2):308–315.

Vijay-Shanker, K. and Joshi, A. (1985). Some computational properties of tree adjoining grammars. In *23rd Meeting of the Association for Computational Linguistics*, pages 82–93, Chicago, Illinois.

Vijay-Shanker, K. and Weir, D. (1990). Polynomial parsing of combinatory categorial grammars. In *28th Meeting of the Association for Computational Linguistics*, pages 1–8, Pittsburgh, PA.

Vijay-Shanker, K. and Weir, D. (1993a). Parsing some constrained grammar formalisms. *Computational Linguistics*, 19(4):591–636.

Vijay-Shanker, K. and Weir, D. (1993b). The use of shared forests in tree adjoining grammar parsing. In *Meeting of the European Chapter of the Association for Computational Linguistics*, pages 384–393, Utrecht, Netherlands.

Vijay-Shanker, K. and Weir, D. (1994). The equivalence of four extensions of context-free grammars. *Mathematical Systems Theory*, 27:511–546.

Vijay-Shanker, K., Weir, D., and Joshi, A. (1987). Characterizing structural descriptions produced by various grammatical formalisms. In *25th Meeting of the Association for Computational Linguistics*.

Weir, D. (1988). *Characterizing Mildly Context-Sensitive Grammar Formalisms*. PhD thesis, University of Pennsylvania, Philadelphia, PA.

Wirén, M. (1992). *Studies in Incremental Natural-Language Analysis*. PhD thesis, Linköping University, Linköping, Sweden.

Younger, D. H. (1967). Recognition of context-free languages in time n^3. *Information and Control*, 10(2):189–208.