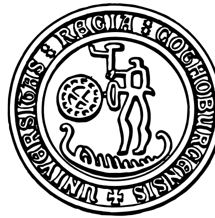
THESIS FOR THE DEGREE OF DOCTOR OF ENGINEERING

The Mechanics of the Grammatical Framework

Krasimir Angelov

CHALMERS | GÖTEBORG UNIVERSITY



Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden

Göteborg, 2011

The Mechanics of the Grammatical Framework
Krasimir Angelov
ISBN 978-91-7385-605-8

© Krasimir Angelov, 2011

Doktorsavhandlingar vid Chalmers tekniska högskola, Ny series nr 3286
ISSN 0346-718X

Technical report 81D
Department of Computer Science and Engineering
Research group: Language Technology
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden
Telephone + 46 (0)31-772 1000

Printed at Chalmers, Göteborg, 2011

Abstract

Grammatical Framework (GF) is a well known theoretical framework and a mature programming language for the description of natural languages. The GF community is growing rapidly and the range of applications is expanding. Within the framework, there are computational resources for 26 languages created from different people in different organizations. The coverage of the different resources varies but there are complete morphologies and grammars for at least 20 languages. This advancement would not be possible without the continuous development of the GF compiler and interpreter.

The demand for efficient and portable execution model for GF has led to major changes in both the compiler and the interpreter. We developed a new low-level representation called Portable Grammar Format (PGF) which is simple enough for an efficient interpretation. Since it was already known that a major fragment of GF is equivalent to Parallel Multiple Context-Free Grammar (PMCFG), we designed PGF as an extension that adds to PMCFG distinctive features of GF such as multilingualism, higher-order abstract syntax, dependent types, etc. In the process we developed novel algorithms for parsing and linearization with PMCFG and a framework for logical reasoning in first-order type theory where the proof search can be constrained by the parse chart.

This monograph is the detailed description of the engine for efficient interpretation of PGF and is intended as a reference for building alternative implementations and as a foundation for the future development of PGF.

Preface

One of my very first programming exercises was a question answering system based on a simple database of canned question-answer pairs. Every time when the user asks a question the program looks up the answer in the database and shows it to the user. Perhaps this is a good exercise for beginners but it is a ridiculous approach to question answering. Still this naïve system is an extreme demonstration of the problems that modern natural language processing systems have.

First of all it is very fragile. If the question is not asked in precisely the same way as it is listed in the database, the system will not be able to produce any answer. Any practical solution should allow variations in the syntax. Although it is possible in principle to include many variations of the same question in the database and in this way to relax the problem, the complete enumeration is usually infinite. A more practical way to model large and even infinite sets of strings is to use grammatical formalisms. In fact, our database is nothing else but a very inefficient implementation of trie. Since the trie is a special kind of finite state automaton, it is also an example of a regular grammar. All modern systems use some kind of grammar in one way or another. By using more rigorous models, it is possible to achieve better and better coverage of what the user understands as a natural language question. Still every grammar is just a computer program and what it can understand is hard coded by the programmer. In the end it will suffer from exactly the same problem as our naïve solution had. Just the degree of illness is different.

The second problem is that our database encodes only very limited fixed knowledge of the world. If something is not explicitly stated as a fact, it will not be available as an answer. In other words, our computer program does not have any reasoning capabilities. We can just as well substitute 'reasoning' with 'computing' but the former clearly indicates that there is some logic involved. The application of mathematical logic in natural language processing has a long tradition.

A lot of logical frameworks have been developed and applied in different context. Still there is no single best choice for every application. The logical reasoning can be implemented by using theorem provers, general purpose logical languages like Prolog, database programming languages like SQL or even more traditional languages like C or Java. In all cases, it serves one and the same purpose - to generalize from a set of plain facts to a system which is able to make its own conclusions. Just like with the first problem, adding logical reasoning does not solve the problem completely. The inference has to be explicitly encoded in the system and a mere mortal can implement only a limited number of rules.

None of the existing natural language systems is even close to what an average human is able to do. Most of the researchers in what used to be called Artificial Intelligence, have given up the idea that they can replicate the brilliant creatures of nature and have focused on the development of applications which are limited but still useful.

This monograph is about the internal engine, the mechanics of one particular system - Grammatical Framework (GF). As the name suggests this is not an end user application but rather a framework which the developers can use to develop applications involving grammars. What the name does not say is that the same framework also has mechanisms for logical reasoning. It might look like the name has failed to mention an important aspect of its nature, but actually it is simply an indication that in GF everything is seen as a grammar or a language. This is not so radical if we remember that the Turing machine is an universal computing machine but at the same time it is a language processor.

GF has exactly the same limitations as any other approach to natural language processing. The language coverage and the reasoning capabilities are limited to what the grammarian has encoded in the grammar. What GF really offers is a programming language specialized for grammar writing. Using the language, the user can concentrate on a particular application while the framework offers a range of processing algorithms for free. The grammar writing itself is also simplified because it is possible to use a library of existing grammars, which frees the user from low-level language features like word order, agreement, clitics and others. These features taken together make it easier to develop grammars which are flexible enough to be used in practical applications.

Here we focus on the internals of the framework. The programming language of GF is intentionally not described except with brief examples whenever it is necessary to make the content self contained. A complete reference and a user guide to GF is available in Ranta [2011], and this is the recommended reference for new GF users. The intended audience of this monograph are advanced users

of GF or researchers in natural language processing who want to have more in-depth knowledge of the framework. Since most of the ideas and the algorithms presented here are general enough, they can be reused in other contexts. In fact, it is completely reasonable to use our engine as a back-end for other frameworks. This makes the volume interesting also for researchers who are developing other systems, not necessary connected to GF.

Acknowledgment

…to my wife Vania and my daughter Siana

It is a long journey from my first programming exercise to Grammatical Framework. I worked with many people and I learned something from everyone.

I started the journey with Dobrinka Radeva, my first teacher in programming, and I am very grateful for all the support that she gave me in the first steps. You made the start easy and entertaining and you encouraged me to go further.

Languages were always an interesting topic for me and I found that formal languages are not less interesting than the natural languages. Boyko Bantchev with his constant passion for exploring new programming languages and for applying different paradigms made me interested in formal languages. GF is after all nothing else but a programming language.

I always like to see programming as an art but it is also a craft which you can apply for living. I worked with Daniel Ditchev for five years on industrial applications and this was an important lesson in what really matters for the craftsmen. It really does not matter how sophisticated a software system is, if it is hard to use or if it does not satisfy the user's requirements. On the artistic side, I always find inspiration in the work of Simon Marlow. While working together, you showed me what is really the difference between craft and art.

Last in historical order but not least in importance, this monograph would not be possible without the support of my PhD supervisor Aarne Ranta. I think that the design of GF as a language and the theory behind it is brilliant. If you had not set the foundation, I would not be able to build on top of it.

Contents

Abstract i

Preface iii

Acknowledgment vii

1 Introduction 1
1.1 GF by Example 5

2 Grammar 13
2.1 PGF definition 14
2.2 GF to PGF translation 18
2.3 Parsing 19
2.3.1 The Idea 21
2.3.2 Deduction Rules 22
2.3.3 A Complete Example 25
2.3.4 Soundness 28
2.3.5 Completeness 29
2.3.6 Complexity 30
2.3.7 Tree Extraction 31
2.3.8 Implementation 33
2.3.9 Evaluation 36
2.4 Online Parsing 36
2.5 Linearization 42
2.5.1 Soundness 44
2.5.2 Completeness 44
2.5.3 Complexity 44
2.6 Literal Categories 45

2.7 Higher-Order Abstract Syntax 49
2.8 Optimizations 55
2.8.1 Common Subexpressions Elimination 56
2.8.2 Dead Code Elimination 60
2.8.3 Large Lexicons 66
2.8.4 Hints for Efficient Grammars 67

3 Reasoning 73
3.1 Computation 82
3.2 Higher-order Pattern Unification 88
3.3 Type Checking 94
3.4 Proof Search 99
3.4.1 Random and Exhaustive Generation 101
3.5 Parsing and Dependent Types 104

4 Frontiers 111
4.1 Statistical Parsing 112
4.2 Montague Semantics 122

5 Conclusion 125

A Portable Grammar Format 127

Chapter 1

Introduction

The ideas behind Grammatical Framework (GF) originated around year 1994 as a grammatical notation [Ranta, 1994] which uses Martin-Löf's type theory [Martin-Löf, 1984] for the semantics of natural language. The idea was further developed at Xerox Research Centre in Grenoble where about four years later the first GF version was released.

Now the framework is more than fifteen years old and it has been successfully used in a number of projects. The implementation itself went through several iterations. The language got a module system [Ranta, 2007] which made it possible to reuse existing grammars as libraries instead of rewriting similar things over and over again. Later this led to the development of the resource grammars library [Ranta, 2009] which is perhaps one of the most distinguishing features of GF from grammar engineering point of view.

In the beginning, the parsing was done by approximation with a context-free grammar, followed by post-processing of the parse trees [Ranta, 2004b]. Later the observation of Ljunglöf [2004] that the GF model is very similar to Parallel Multiple Context-Free Grammar (PMCFG) made it possible to develop new parsing algorithms [Ljunglöf, 2004][Burden and Ljunglöf, 2005] that are more efficient and that need less post-processing since they operate on a representation that is semantically closer to the original grammar. Still the algorithms did not operate directly with PMCFG but with a weaker form known as Multiple Context-Free Grammar (MCFG), where the gap between PMCFG and MCFG was filled in by using a form of unification. The new algorithms soon superseded the original context-free parser but still for large grammars and morphologically rich languages the parser was a bottleneck. The current GF engine uses a new algorithm [Angelov, 2009] which is further optimized and for some languages it leads to a speed-up

of several orders of magnitude. For instance an experiment with the resource grammars library shows that while for English the efficiency is nearly the same, for German the difference is about 400 times. For other languages like Finnish and all Romance languages the difference is not even measurable because the old parser quickly exceeds the available memory. All this scalability issues, however, are apparent only on the scale of the resource grammars while for small application grammars they are insignificant. Since originally the resource grammars were designed only as a tool for deriving application grammars and not as grammars for parsing, this was never considered as an important performance issue. For instance both the Finnish and the Romance resource grammars were successfully used as libraries in different applications. The main improvement is that now these grammars can be used directly for parsing which permits the development of applications with wider coverage.

The measurable speed-up is by wall-clock time but still the theoretical complexity stays the same. In other words, the new algorithm is quicker in analysing commonly occurring syntactic patterns, but in principle it is still with polynomial complexity and in extreme cases the exponent can be very high. Fortunately, such pathological cases does not occur in natural languages and our empirical studies show that at least for the resource grammar library the complexity is linear.

The new algorithm has also other advantages. First of all, it naturally supports PMCFG rather than some weaker formalism like MCFG. When later the PMCFG formalism was extended with literals and higher-order syntax this became the first model which fully covers the semantics of GF without the need for pre- or post-processing. Another advantage is that the parser is incremental. This made it possible to develop specialized user interfaces which help the users in writing grammatically correct content in a controlled language, i.e. in a subset of some natural language [Bringert et al., 2009] [Angelov and Ranta, 2010]. In such a scenario, the user has to be aware of the limitations of the grammar, and he is helped by the interface which shows suggestions in the style of the T9 interface for mobile phones. Since the suggestions are computed from the incremental parser this ensures that the content is always in the scope of the grammar. While similar interface can be build by using an incremental parser for context-free grammars [Earley, 1970], it cannot achieve the same goal since it can work only with an approximation of the original grammar. All of the actively developed user interfaces for GF are based on the new incremental parser but here my personal involvement is more modest and a lot of work was also done by Björn Bringert, Moisés Salvador Meza Moreno, Thomas Hallgren, Grégoire Détrez and Ramona Enache.

The parsing algorithm together with its further refinements is perhaps the most

central contribution of this monograph but it is not the end, since we are aiming at a platform that is easy to use in practical applications. The further development of GF demanded better separation between compiler and interpreter and in Angelov et al. [2008], we developed the first version of the Portable Grammar Format (PGF) which is a representation that is simple enough for efficient interpretation by a relatively small runtime engine. This joint work with Björn Bringert and Aarne Ranta was the first solution that allowed the distribution of standalone GF applications. Unfortunately, it had the disadvantage that there were two different grammar representations. The first is more compact but can be used only for linearization, and the second is basically an extension of PMCFG that is used only for parsing. Although later it became clear that linearization is also possible with the second representation, at that time it was still necessary to have them both since the PMCFG representation is usually big, so for applications that do not use the parser we can generate only the first representation. Fortunately, the difference was substantially reduced after the development of grammar optimization techniques, so soon the first representation was completely dropped. The new incarnation of PGF is much simpler and now a completely different linearization algorithm is used, so the original design became obsolete and this monograph is currently the only up-to-date reference.

In fact, the simplicity of the engine made it possible to reimplement it in five different languages - Haskell, JavaScript, Java, C# and C. The implementation in Haskell is still the only one that is complete and this is solely my personal contribution. The other implementations are credited to Björn Bringert, Moisés Salvador Meza Moreno and myself for JavaScript, Grégoire Détrez and Ramona Enache for Java [Enache and Détrez, 2010], Christian Ståhlfors and Erik Bergström for C# and Lauri Alanko for C.

The algorithms for parsing and linearization in the PGF engine are the main topics of the second chapter of this monograph but this is only half of the way to a complete GF implementation. The third chapter is a reference to the algorithms for evaluation, unification, type checking, and proof search which taken together realize the logical aspects of the language. Most of the algorithms in this last chapter are not new but we felt that a reference to the PGF engine cannot be complete without a detailed description of those, since otherwise the reader will have to be redirected to a long list of other sources which furthermore describe only small fragments, and still the composition of the whole puzzle may be quite tricky. More concretely the design of the GF logic was influenced by the work of Ulf Norrel on Agda [Norell, 2007] and the work of Xiaochu Qi on λProlog [Qi, 2009]. Although GF has its logical framework right from the beginning, it

was not actively developed, while there was already a lot of interesting research in dependently typed functional languages (Agda) and logical languages (λProlog), so when the PGF engine was designed, we decided to simply take the best that suits our needs. The real contribution of this last chapter is at the end where we show how the logical framework of GF is integrated with the parser and this let us to impose complex semantic restrictions in the grammar.

The development of the PGF engine is an important milestone in the evolution of the framework, and this monograph is a complete reference to all details of the mechanics behind it. Our main focus is on the algorithms, but in Appendix A we also include the exact description the portable grammar format as it is in GF 3.2. What we do not describe, however, is the GF language itself, since this can be found in many other sources. In particular, the reader is referred to Ranta [2011] for complete reference. Still to make the monograph self-contained, in the next section, we will introduce the framework by brief examples which illustrate the main points.

There are exciting new developments that unfortunately we had to separate from the main content because this is still the front line of the research around GF. From one side, the advancement of the logical aspects in the framework makes it possible to embed formal ontologies in the grammars. The best methodology however is still far from clear. From another side, the improvement in the parsing performance opens the way for bridging the gap between controlled languages and open-domain text. Still preliminary study shows that the current English resource grammar combined with Oxford Advanced Learner's Dictionary can cover up to 91.75% of the syntactic constructions found in Penn Treebank [Marcus et al., 1993]. Although this is a promising start there are still two problems to be solved. First, the parser has to be more robust and it should not fail when it is faced with some of the unknown constructions in the remaining 8.25%. Second, a statistical disambiguation model is needed since the resource grammars are highly ambiguous. Although PGF includes a simple probabilistic model, it will have to be extended in order to scale up to the complexity of Penn Treebank. This new research goals are not fully accomplished yet, but we felt that it is still worth to explore the frontiers, and we devoted the fourth chapter to the possible solutions for this two problems.

1.1 GF by Example

GF is a Logical Framework in the spirit of Harper et al. [1993] extended with a framework for defining concrete syntax for the realization of the formal meanings as natural language expressions. Every GF grammar has one abstract syntax defined in the Logical Framework and one or more concrete syntaxes. The abstract syntax is the abstract theory (the ontology) of the particular application domain while the concrete syntax is language-dependent and reflects the syntax and the pragmatics of some specific language. The definitions in the concrete syntax are reversible which makes it possible to use the grammar for both parsing and linearization (generation). Since it is allowed to have many concrete syntaxes attached to the same abstract syntax, the abstract syntax can be used as a translation interlingua between different languages.

The logical framework of the abstract syntax is Martin-Löf's type theory. It consists of a set of definitions for basic types and functions. For example, the Foods grammar from the GF tutorial (Chapter 2 in Ranta [2011]) has the following basic types:

$$\mathbf{cat}\ \mathit{Phrase};\ \mathit{Item};\ \mathit{Kind};\ \mathit{Quality};$$

In GF, the basic types play the role of abstract syntactic categories. Since we have not introduced the concrete syntax yet, for now they are just names for us. Because of the duality between basic types and abstract categories, often we will use them as synonyms but when we want to emphasise the logical aspect of the framework then we will say type and when we talk about the syntactic aspect we will say category.

In our simple grammar, we can talk about only four kinds of food and they are all defined as constants (functions without arguments) of type *Kind*:

$$\mathbf{fun}\ \mathit{Wine}, \ \mathit{Cheese}, \ \mathit{Fish}, \ \mathit{Pizza}\ :\ \mathit{Kind};$$

Once we have the different kinds we need a way to point to this or that piece of food. We define four more functions which act like determiners:

$$\mathbf{fun}\ \mathit{This}, \mathit{That}, \mathit{These}, \mathit{Those} : \mathit{Kind} \rightarrow \mathit{Item};$$

Note that all functions take as an argument some general kind and return one particular item of food of the same kind. Grammatically *this* and *that* are determiners but from the logical point of view they are just functions.

Similarly to the kinds we can also introduce different food qualities as constants:

$$\textbf{fun}\ \mathit{Fresh}, \mathit{Warm}, \mathit{Delicious} : \mathit{Quality};$$

Note that in both cases these are just constants which grammatically will correspond to single words. However logically they play very different roles, so to distinguish we assign different types to them. The combination of kinds, determiners and qualities let us to express opinions about a concrete item:

$$\textbf{fun}\ \mathit{Is} : \mathit{Item} \rightarrow \mathit{Quality} \rightarrow \mathit{Phrase};$$

For instance, the opinion that “this pizza is delicious” can be encoded as the abstract expression:

$$\mathit{Is}\ (\mathit{This}\ \mathit{Pizza})\ \mathit{Delicious}$$

So far there was nothing surprising. We just defined some types and functions and by applying these functions we constructed expressions which encoded particular logical facts. Now we want to generate natural language and for this we have to introduce the concrete syntax in GF. The concrete syntax is nothing else but the implementation of the abstract types and functions in some natural language.

We start with English because as usual the English implementation is the simplest one. In the concrete syntax, every abstract category corresponds to some implementation type. The implementation for *Kind*:

$$\textbf{lincat}\ \mathit{Kind} = \{s : \mathit{Number} \Rightarrow \mathit{Str}\};$$

is a record with one field ’s’ which is a table indexed by the parameter *Number*. We need the table in order to handle the inflection in plural i.e. we can generate either “this pizza” or “these pizzas”. An example implementation of *Pizza* is:

$$\textbf{lin}\ \mathit{Pizza} = \{s = \textbf{table}\ \{\mathit{Sg} \Rightarrow \text{"pizza"}; \mathit{Pl} \Rightarrow \text{"pizzas"}\}\};$$

Now by selecting from the table the element indexed by *Sg* we get the singular form ”pizza” and by selecting *Pl* we get the plural ”pizzas”.

Before proceeding with the determiners we have to define the linearization type of *Item*:

$$\textbf{lincat}\ \mathit{Item} = \{s : \mathit{Str}; n : \mathit{Number}\};$$

Again we have a record with field s but this time we also have the second field *n* which is the grammatical number of the item. This time the s field is not a table because when we apply the determiner the number will be fixed, so we need only one form. We added the second field because in the implementation of function *Is* we will have to know whether the item is in singular or plural in order to choose the right form of the copula i.e. "is" or "are". The implementation of the determiner *This* fixes the number to be singular while *These* chooses plural:

$$\begin{aligned} \textbf{lin}\ \textit{This}\ \ k &= \{s = \text{"this"} +\!\!+\ k.s\ !\ \textit{Sg}; n = \textit{Sg}\}; \\ \textit{These}\ k &= \{s = \text{"these"} +\!\!+\ k.s\ !\ \textit{Pl}; n = \textit{Pl}\}; \end{aligned}$$

For the qualities we do not need anything else except the corresponding English word. The definitions of the category and the functions are pretty trivial:

$$\begin{aligned} \textbf{lincat}\ \textit{Quality} &= \{s : \textit{Str}\}; \\ \textbf{lin}\ \textit{Fresh} &= \{s = \text{"fresh"}\}; \\ \textit{Warm} &= \{s = \text{"warm"}\}; \\ \textit{Delicious} &= \{s = \text{"delicious"}\}; \end{aligned}$$

As we said before, for the linearization of the function *Is* we need to know whether the item is in singular or in plural. The linearization type of *Phrase* and the implementation of *Is* are defined as:

$$\begin{aligned} &\textbf{lincat}\ \textit{Phrase} = \{s : \textit{Str}\}; \\ &\quad \textbf{lin}\ \textit{Is}\ i\ q = \{s = i.s +\!\!+\ \textbf{case}\ i.n\ \textbf{of}\ \{\textit{Sg} \Rightarrow \text{"is"}; \textit{Pl} \Rightarrow \text{"are"}\} +\!\!+\ q.s\}; \end{aligned}$$

Here we check the number by pattern matching on the value of $i.n$ and this lets us to select the right form of the copula.

An important feature in the GF grammar model is that all language dependent constructions are encoded in the concrete syntax rather than in the abstract. This allows the abstract syntax to be made purely semantic. For instance in the English version of the Foods grammar the choice of the words, the inflection forms and the number agreement are encoded in the concrete syntax. Exactly the same abstract syntax can be reused for other natural languages. As an example, Figure 1.1 contains the concrete syntax for the same grammar in Bulgarian. Although this is a sufficiently different language it still fits quite well in the same abstract syntax. In Bulgarian, the words agree not only in number but also in gender when the noun (the kind) is in singular. As you can see, now the implementation of the category

```
param Gender = Masc | Fem | Neutr;
      Number = Sg | Pl;
      Agr = ASg Gender | APl;

lincat Phrase = {s : Str};
       Quality = {s : Agr ⇒ Str};
       Item = {s : Str; a : Agr};
       Kind = {s : Number ⇒ Str; g : Gender};

lin Is i q = i.s ++ case i.a of {ASg _ ⇒ "e"; APl ⇒ "sa"} ++ q.s ! i.a;
    This k = {s = case k.g of {Masc ⇒ "tozi"; Fem ⇒ "tazi"; Neutr ⇒ "tova"} ++ k.s ! Sg; a = ASg k.g};
    These k = {s = "tezi" ++ k.s ! Pl; a = APl};
    Wine = {s = table {Sg ⇒ "vino"; Pl ⇒ "vina"}; g = Neutr};
    Cheese = {s = table {Sg ⇒ "sirene"; Pl ⇒ "sirena"}; g = Neutr};
    Fish = {s = table {Sg ⇒ "riba"; Pl ⇒ "ribi"}; g = Fem};
    Pizza = {s = table {Sg ⇒ "pica"; Pl ⇒ "pici"}; g = Fem};
    Fresh = {s = table {ASg Masc ⇒ "svež"; ASg Fem ⇒ "sveža";
                        ASg Neutr ⇒ "svežo"; APl ⇒ "sveži"}};
    Warm = {s = table {ASg Masc ⇒ "gorešt"; ASg Fem ⇒ "gorešta";
                       ASg Neutr ⇒ "gorešto"; APl ⇒ "gorešti"}};
    Delicious = {s = table {ASg Masc ⇒ "prevăzhoden"; Fem ⇒ "prevăzhodna";
                            ASg Neutr ⇒ "prevăzhodno"; APl ⇒ "prevăzhodni"}};
```

Figure 1.1: The Foods grammar for Bulgarian

Kind has one more field in the record which contains the grammatical gender. The category *Item* which in English had an extra field n for the number now has a field a of type *Agr* which encodes both the number and the gender when the word is in singular. The linearization of the quality should agree in number and gender with the kind so in the new implementation of *Quality* the field s is now a table indexed by *Agr* instead of a plain string.

It is an important observation that the concrete syntax is all about the manipulation of tuples of strings i.e. tables and records. The tuples are a key feature in the PMCFG formalism so it is not surprising that GF is reducible to PMCFG. Chapter 2 explains how PMCFG is used for parsing and natural language generation.

So far we have used only simple types in the abstract syntax. It is always a good idea to keep the syntax as simple as possible but sometimes we want to put even more semantics and then the simple types are not sufficient anymore. The

abstract syntax is a complete logical framework and we can do arbitrary computations and logical inferences in it. As an illustrative example, we can extend the foods grammar with measurement expressions. We want to say things like "two bottles of wine" or "one litre of wine" but we do not want to allow "two pieces of wine". It should still be possible to ask for "two pieces of pizza". The allowed metrical units are dependent on the particular kind of food.

First we have to define a set of measurement units that we can use. We add a category *Unit* and some constants in the grammar:

$$\textbf{cat}\ \mathit{Unit};$$
$$\textbf{fun}\ \mathit{Bottle}, \mathit{Litre}, \mathit{Kilogram}, \mathit{Piece} : \mathit{Unit};$$

The allowed combinations of *Kind* and *Unit* can be specifyed by having some logical predicate which is true only for the valid combinations. In type theory, the logical propositions are identified with the types, so our predicate is just yet another category:

$$\textbf{cat}\ \mathit{HasMeasure}\ \mathit{Kind}\ \mathit{Unit};$$

The new thing is that now the category is not just a name but it also has two indices - one of type *Kind* and one of type *Unit*. Every time when we use the category we also have to give concrete values for the indices. For example, the way to specify the allowed units for every kind is to add one constant of category *HasMeasure* for every valid combination, where the category is indexed by the right values:

$$\begin{aligned}\textbf{fun}\ & \mathit{wine_bottle} : \mathit{HasMeasure}\ \mathit{Wine}\ \mathit{Bottle};\\ & \mathit{cheese_kilogram} : \mathit{HasMeasure}\ \mathit{Cheese}\ \mathit{Kilogram};\\ & \mathit{fish_kilogram} : \mathit{HasMeasure}\ \mathit{Fish}\ \mathit{Kilogram};\\ & \mathit{pizza_piece} : \mathit{HasMeasure}\ \mathit{Pizza}\ \mathit{Piece};\end{aligned}$$

We can connect this new definitions with the other parts of the grammar by providing a function which constructs an item consisting of a certain number of units:

$$\textbf{fun}\ \mathit{NumItem} : \mathit{Number} \to (u : \mathit{Unit}) \to (k : \mathit{Kind}) \to \mathit{HasMeasure}\ k\ u \to \mathit{Item};$$

We ensured that only valid units are allowed by adding an extra argument of category *HasMeasure*. The category is indexed by k and u, and the notations $(u : \mathit{Unit})$ and $(k : \mathit{Kind})$ mean that these are exactly the values of the second and the third arguments of function *NumItem*.

For instance the phrase "two bottles of wine" is allowed and has the abstract syntax[1]:

$$\mathit{NumItem}\ 2\ \mathit{Bottle}\ \mathit{Wine}\ \mathit{wine_bottle}$$

The phrase "two pieces of wine" is not allowed because there is no way to construct an expression of type *HasMeasure Wine Piece*.

This extra argument is purely semantic and is not linearized in the concrete syntax. The linearization of *NumItem* can be defined as:

$$\textbf{lin}\ \mathit{NumItem}\ n\ u\ k\ _ = \{s = n.s \mathbin{++} u.s\ !\ n.n \mathbin{++} \text{"of"} \mathbin{++} k.s\ !\ \mathit{Sg}\};$$

Here the last argument is not used at all in the linearization and we do not even give a name for it. Instead we use the wildcard symbol '_'. Still if we parse "two bottles of wine", the parser correctly fills in the argument with *wine_bottle*.

The magic here is that the parser is integrated with a type checker and a theorem prover. There are three steps in the parsing. The first step is purelly syntactic and at this step the parser recovers as much details for the abstract syntax as possible based only on the input string. In the second step, the partial abstract syntax tree is type checked to verify that there are no semantic violations. At this step, the type checker is already able to fill in some holes based only on the type constraints. However, in our case this is not possible because the output after the second step will be:

$$\mathit{NumItem}\ 2\ \mathit{Bottle}\ \mathit{Wine}\ ?$$

where the question mark $?$ is a place holder for missing information. The type checker cannot fill in the hole but at least it is able to determine that it should be filled in with something of type *HasMeasure Wine Bottle*. This type is used as a goal for the theorem prover. For all holes, left in the tree after the type checking, the theorem prover tries to find a proof that there is an expression of this type. If the search is successful, the hole is replaced with the found value.

The proof search can be arbitrarily complex because we can also add inference rules. An inference rule in GF is nothing else but yet another function. For instance, if we want to say that everything that is measurable in bottles is also measurable in litres we can add:

$$\textbf{fun}\ \mathit{to_Litre} : (k : \mathit{Kind}) \rightarrow \mathit{HasMeasure}\ k\ \mathit{Bottle} \rightarrow \mathit{HasMeasure}\ k\ \mathit{Litre};$$

[1] Strictly speaking we need something more complex for the numeral "two". The GF resource library provides abstract syntax for numerals but here for simplicity we just write the number 2.

Now the proof that wine is measurable in litres is the term:

$$\langle \mathit{to_Litre}\ \mathit{Wine}\ \mathit{wine_bottle} : \mathit{HasMeasure}\ \mathit{Wine}\ \mathit{Litre} \rangle$$

The theorem prover is not an internal component of the parser. It can be invoked directly by the user and in this way the GF grammar can be used as a static knowledge base. For instance in the GF shell the user can issue the query "Is the wine measurable in litres?" by using the exhaustive generation command (`gt`):

```
> gt -cat="HasMeasure Wine Litre"
to_Litre Wine wine_bottle
```

Since the proof in GF for any theorem is just an abstract syntax tree, we can just as well linearize it. For example, if we want to see the above proof in natural language, then we can add the linearization rules:

lincat *HasMeasure* $= \{s : \mathit{Str}\}$;
lin *wine_bottle* $= \{s =$ "wine is measurable in bottles"$\}$;
to_Litre $k\ m = \{s =$ "wine is measurable in litres because" $+\!\!+\ m.s\}$;

Now we can pipe the exhaustive generation command with the linearization command:

```
> gt -cat="HasMeasure Wine Litre" | l
wine is measurable in litres
because wine is measurable in bottles
```

and we will see the proof rendered in English.

Detailed explanations of the design of the type checker and the theorem prover and the interaction between natural language and logic are included in Chapter 3.

Chapter 2

Grammar

The language of the GF concrete syntax is elegant and user friendly but too complex to be suitable for direct machine interpretation. Fortunately Ljunglöf [2004] identified Parallel Multiple Context-Free Grammar (PMCFG) [Seki et al., 1991] as a suitable low-level representation for the concrete syntax in GF.

PMCFG is one of the formalisms that have been proposed for the syntax of natural languages. It is an extension of Context-Free Grammar (CFG) where the right-hand side of the production rule is a tuple of strings instead of only one string. The generative power and the parsing complexity of PMCFG and the closely related MCFG formalism has been thoroughly studied in Seki et al. [1991], Seki et al. [1993] and Seki and Kato [2008]. Using tuples the formalism can model discontinuous constituents which makes it more powerful than CFG. Its expressiveness also subsumes other well known formalisms like Tree Adjoining Grammars [Joshi et al., 1975] and Head Grammars [Pollard, 1984]. The discontinuity is also the key feature which makes it suitable as an assembly language for GF. At the same time, PMCFG has the advantage to be parseable in polynomial time which is computationally attractive. Different algorithms for parsing with MCFG are presented in Nakanishi et al. [1997], Ljunglöf [2004] and Burden and Ljunglöf [2005]. None of them, however, covers the full expressivity of PMCFG so we developed our own algorithm [Angelov, 2009].

Here we do not want to repeat the details in Ljunglöf's algorithm for compiling the concrete syntax in GF to PMCFG. The algorithm is part of the GF compiler, which is not our main topic. Still a basic intuition for the relation between GF and PMCFG can help the reader to understand the mechanics of the GF engine. Furthermore, we have to add a representation for the abstract syntax in order to represent a complete GF grammar. The abstract syntax and PMCFG together are

the main building blocks of the Portable Grammar Format (PGF) which is our runtime grammar representation.

We will formally define the notion of PGF and PMCFG in Section 2.1 and after that, in Section 2.2, we will give the basic idea of how the GF source code is compiled to it. The remaining sections are the main content of this chapter and there we present the rules for parsing and natural language generation with PGF. We start with simplified rules which cover only grammars with context-free abstract syntax but after that we generalize to literal categories and higher-order abstract syntax. In the last section we also introduce some automatic and manual techniques for grammar optimization.

2.1 PGF definition

This section is the formal definition of PGF. It is useful as a reference but it is not necessary to remember all the details from the first reading. We advice the reader to scan quickly through the content and come back to it later if some notation in the next sections is not clear.

Definition 1 *A grammar in* ***Portable Grammar Format (PGF)*** *is a pair of an abstract syntax* $\mathcal{A}$ *and a finite set of concrete syntaxes* $\mathcal{C}_1, \ldots, \mathcal{C}_n$*:*

$$\mathcal{G} \;=< \mathcal{A}, \{\mathcal{C}_1, \ldots, \mathcal{C}_n\} >$$

Definition 2 *An* ***abstract syntax*** *is a triple of a set of abstract categories, a set of abstract functions with their type signatures and a start category:*

$$\mathcal{A} \;=< N^{\mathcal{A}}, F^{\mathcal{A}}, S >$$

- $N^{\mathcal{A}}$ *is a finite set of abstract categories.*
- $F^{\mathcal{A}}$ *is a finite set of abstract functions. Every element in the set is of the form* $f : \tau$ *where* f *is a function symbol and* τ *is its type. The type is either a category* $C \in N^{\mathcal{A}}$ *or a function type* $\tau_1 \rightarrow \tau_2$ *where* τ_1 *and* τ_2 *are also types*[1]*. Overloading is not allowed, i.e. if* $f : \tau_1 \in F^{\mathcal{A}}$ *and* $f : \tau_2 \in F^{\mathcal{A}}$ *then* $\tau_1 = \tau_2$*.*
- $S \in N^{\mathcal{A}}$ *is the start category.*

[1] In Section 3 we will extend the notion of types to dependent types

Definition 3 *A* ***concrete syntax*** $\mathcal{C}$ *is a Parallel Multiple Context-Free Grammar complemented with a mapping from its categories and functions into the abstract syntax:*

$$\mathcal{C} \ = < G, \psi_N, \psi_F, d >$$

- *G is a Parallel Multiple Context-Free Grammar*
- *ψ_N is a mapping from the concrete categories in G to the set of abstract categories $N^{\mathcal{A}}$.*
- *ψ_F is a mapping from the concrete functions in G to the set of abstract functions $F^{\mathcal{A}}$.*
- *d assigns a positive integer $d(A)$, called dimension, to every abstract category $A \in N^{\mathcal{A}}$. One and the same category can have different dimensions in the different concrete syntaxes.*

PMCFG is a simple extension of CFG where every syntactic category is defined not as a set of strings but as a set of tuples of strings. We get a tuple in one category by applying a function over tuples from other categories.

For the definition of functions in PMCFG it is useful to introduce the notion of arity. The arity of an abstract function $f^{\mathcal{A}}$ is the number of arguments $a(f^{\mathcal{A}})$ that it takes. The arity can be computed from the type of the function:

$$a(f^{\mathcal{A}}) = a(\tau), \qquad \text{if } f^{\mathcal{A}} : \tau \in F^{\mathcal{A}}$$

where the arity of the type $a(\tau)$ is computed by counting how deeply the function type is nested to the right:

$$a(\tau) = \begin{cases} 0, & \tau \equiv C \text{ , where } C \in N^{\mathcal{A}}, \\ 1 + a(\tau_2), & \tau \equiv \tau_1 \rightarrow \tau_2 \text{ , where } \tau_1, \tau_2 \text{ are types} \end{cases}$$

Since in the concrete syntax there is a mapping from every concrete function to the corresponding abstract function we can also transfer the notion of arity to the concrete syntax. The arity of a concrete function $f^{\mathcal{C}}$ is:

$$a(f^{\mathcal{C}}) = a(\psi_F(f^{\mathcal{C}})), \qquad \text{if } f^{\mathcal{C}} \text{ is a concrete function}$$

For the definitions of concrete functions itself, we use a notation which is a little bit unconventional but this will make it easier to write deduction rules later. An example of a function is:

$$f := (\langle 1; 1 \rangle b, \langle 2; 1 \rangle \langle 1; 2 \rangle)$$

Here f is the function name. It creates a tuple of two strings where the first one $\langle 1;1\rangle b$ is constructed by taking the first constituent of the first argument and adding the terminal b at the end. The second one $\langle 2;1\rangle\langle 1;2\rangle$ concatenates the first constituent of the second argument with the second constituent of the first argument. In general, the notation $\langle d;r\rangle$ stands for argument number d and constituent number r.

The grammar itself is a set of productions which define how to construct a given category from a list of other categories by applying some function. An example using function f is the production:

$$A \rightarrow f[B, C]$$

Now the following is the formal definition of a PMCFG:

Definition 4 *A **parallel multiple context-free grammar** is a 5-tuple:*

$$G = < N^{\mathcal{C}}, F^{\mathcal{C}}, T, P, L >$$

- $N^{\mathcal{C}}$ *is a finite set of concrete categories. The equation* $d(A) = d(\psi_N(A))$ *defines the dimension for every concrete category as equal to the dimension in the current concrete syntax of the corresponding abstract category.*

- $F^{\mathcal{C}}$ *is a finite set of concrete functions where the dimensions* $r(f)$ *and* $d_i(f)$ $(1 \leq i \leq a(f))$ *are given for every* $f \in F^{\mathcal{C}}$. *For every positive integer* d, $(T^*)^d$ *denotes the set of all* d*-tuples of strings over* T. *Each function* $f \in F^{\mathcal{C}}$ *is a total mapping from* $(T^*)^{d_1(f)} \times (T^*)^{d_2(f)} \times \cdots \times (T^*)^{d_{a(f)}(f)}$ *to* $(T^*)^{r(f)}$, *and is defined as:*

$$f := (\alpha_1, \alpha_2, \ldots, \alpha_{r(f)})$$

  Here α_i *is a sequence of terminals and* $\langle k;l\rangle$ *pairs, where* $1 \leq k \leq a(f)$ *is called argument index and* $1 \leq l \leq d_k(f)$ *is called constituent index. Sometimes we will use the notation rhs*(f, l) *to refer to constituent* α_l *of* f.

- T *is a finite set of terminal symbols.*

- P *is a finite set of productions of the form:*

$$A \rightarrow f[A_1, A_2, \ldots, A_{a(f)}]$$

  where $A \in N^{\mathcal{C}}$ *is called result category,* $A_1, A_2, \ldots, A_{a(f)} \in N^{\mathcal{C}}$ *are called argument categories and* $f \in F^{\mathcal{C}}$ *is a function symbol. For the production to be well formed the conditions* $d_i(f) = d(A_i)$ $(1 \leq i \leq a(f))$ *and* $r(f) = d(A)$ *must hold.*

- $L \subset N^{\mathcal{C}} \times F^{\mathcal{C}}$ *is a set which defines the default linearization functions for those concrete categories that have default linearizations. If the pair* (A, f) *is in* L *then* f *is a default linearization function for* A*. We will also use the abbreviation:*

$$\textit{lindef}(A) = \{f \mid (A, f) \in L\}$$

to denote the set of all default linearization functions for A*. For every* $f \in$ *lindef*(A) *it must hold that* $r(f) = d(A)$*,* $a(f) = 1$ *and* $d_1(f) = 1$*.*

We use similar definition of PMCFG as the one used by Seki and Kato [2008] and Seki et al. [1993] except that they use variable names like x_{kl} while we use $\langle k; l\rangle$ to refer to the function arguments. We also defined default linearization functions which are used in the linearization of incomplete and higher-order abstract syntax trees.

The abstract syntax of the grammar defines some function types which let us construct typed lambda terms. Although this is not visible for the user of the PGF format, the same is possible with the concrete syntax. We can combine functions from the PMCFG grammar to build concrete syntax trees. The concrete trees are formally defined as:

Definition 5 $(f\ t_1 \ldots t_{a(f)})$ *is a concrete tree of category* A *if* t_i *is a concrete tree of category* B_i *and there is a production:*

$$A \rightarrow f[B_1 \ldots B_{a(f)}]$$

The abstract notation for "t *is a tree of category* A*" is* $t : A$*. When* $a(f) = 0$ *then the tree does not have children and the node is called a leaf.*

Once we have a concrete syntax tree we can linearize it in a bottom-up fashion to a string or a tuple of strings. The functions in the leaves of the tree do not have arguments so the tuples in their definitions already contain constant strings. If the function has arguments, then they have to be linearized and the results combined. Formally this can be defined as a function $\mathcal{L}$ applied to the concrete tree:

$$\begin{aligned}
&\mathcal{L}(f\ t_1\ t_2 \ldots t_{a(f)}) = (x_1, x_2 \ldots x_{r(f)}) \\
&\quad \text{where } x_i = \mathcal{K}(\mathcal{L}(t_1), \mathcal{L}(t_2) \ldots \mathcal{L}(t_{a(f)}))\ \alpha_i \\
&\quad \text{and } f := (\alpha_1, \alpha_2 \ldots \alpha_{r(f)}) \ \in \ F^{\mathcal{C}}
\end{aligned}$$

The function uses a helper function $\mathcal{K}$ which takes the vector of already linearized arguments and a sequence α_i of terminals and $\langle k; l \rangle$ pairs and returns a string. The string is produced by substitution of each $\langle k; l \rangle$ with the string for constituent l from argument k:

$$\mathcal{K}\ \vec{\sigma}\ (\beta_1 \langle k_1; l_1 \rangle \beta_2 \langle k_2; l_2 \rangle \ldots \beta_n) = \beta_1 \sigma_{k_1 l_1} \beta_2 \sigma_{k_2 l_2} \ldots \beta_n$$

where $\beta_i \in T^*$. The recursion in $\mathcal{L}$ terminates when a leaf is reached.

2.2 GF to PGF translation

In GF, we define abstract functions in the abstract syntax and corresponding linearization rules in the concrete syntax. In PGF, the abstract functions are preserved but the linearization rules are replaced with concrete functions. In general, there is a many to one mapping between the concrete and the abstract functions, because the compilation of every linearization rule leads to the generation of one or more concrete functions. In a similar way, the linearization types for the abstract categories are represented as a set of concrete categories. The relation between abstract and concrete syntax is preserved by the mappings ψ_N and ψ_F which map concrete categories to abstract categories and concrete functions to abstract functions.

The main differences between the Parallel Multiple Context-Free Grammar in PGF and the concrete syntax of GF are that the former allows only flat tuples instead of nested records and tables, and that PMCFG does not allow parameters while GF does. The nested records and the tables are easy to implement in PMCFG by flattening the nested structures. The parameters however are more tricky and this is the main reason for the many to one relation between concrete and abstract syntax. Instead of explicitly passing around parameters during the execution, we instantiate all parameter variables with all possible values and from the instantiations we generate multiple concrete functions and categories.

If we take as an example the linearization type for category *Item* (Chapter 1):

$$\textbf{lincat}\ \mathit{Item} = \{s : \mathit{Str}; n : \mathit{Number}\};$$

then in PMCFG, *Item* will be split into two categories - one for singular and one for plural[2]:

$$\mathit{Item}_{\mathit{Sg}},\ \mathit{Item}_{\mathit{Pl}}$$

[2] In the real compiled code, all concrete categories and functions are just integers but here we give them mnemonic names to make the examples more readable.

The functions are multiplied as well. For example we will generate two productions from function *Is*:

$$Phrase \rightarrow Is_{Sg}[Item_{Sg}, Quality]$$
$$Phrase \rightarrow Is_{Pl}[Item_{Pl}, Quality]$$

where every production uses a different concrete function:

$$Is_{Sg} := (\langle 1; 1\rangle \text{ "is" } \langle 2; 1\rangle)$$
$$Is_{Pl} := (\langle 1; 1\rangle \text{ "are" } \langle 2; 1\rangle)$$

We do not need parameters because the inflection is guided by the choice of the function. If we use Is_{Sg}, we will get the word "is" and if we use Is_{Pl}, then we will get "are". The relation between abstract and concrete syntax is kept in the mappings ψ_N and ψ_F which in our example are:

$$\psi_N(Item_{Sg}) = Item \qquad \psi_F(Is_{Sg}) = Is$$
$$\psi_N(Item_{Pl}) = Item \qquad \psi_F(Is_{Pl}) = Is$$

2.3 Parsing

The parser in the GF engine is described separately in Angelov [2009]. This section is an extended version of the same paper and here we are more explicit about how the parser fits in the engine. The algorithm has two advantages compared to the algorithms [Ranta, 2004b][Ljunglöf, 2004][Burden and Ljunglöf, 2005] used in GF before - it is more efficient and it is incremental.

The incrementality means that the algorithm reads the input one token at a time and calculates all possible continuations, before the next token is read. There is a substantial evidence showing that humans process language in an incremental fashion which makes the incremental algorithms attractive from a cognitive point of view.

Our algorithm is also top-down which makes it possible by using the grammar to predict the next word from the sequence of preceding words. This is used for example in text based dialog systems or authoring tools for controlled languages [Angelov and Ranta, 2010] where the user might not be aware of the grammar coverage. With the help of the parser, the authoring tool (Figure 2.1) suggests the possible continuations and in this way the user is guided for how to stay within the scope of the grammar. The tool also highlights the recognized phrases ("switch"

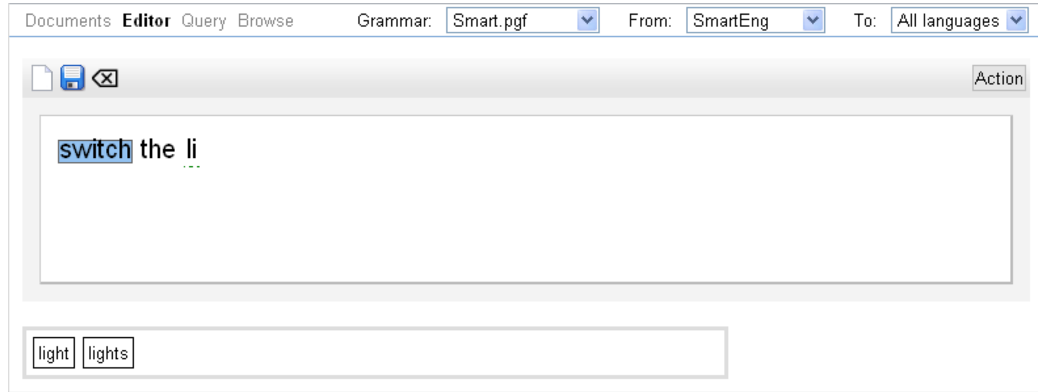


Figure 2.1: An authoring tool guiding the user to stay within the scope of the controlled language

on the figure) and this is possible even before the sentence is complete since the parser is able to produce the parse tree incrementally.

In this section, in order to illustrate how the parsing works, we will use as a motivating example the $a^n b^n c^n$ language which in PMCFG is defined as:

$$
\begin{aligned}
S &\rightarrow c[N] \\
N &\rightarrow s[N] \\
N &\rightarrow z[]
\end{aligned}
$$

$$
\begin{aligned}
c &:= \ (\langle 1;1\rangle\ \langle 1;2\rangle\ \langle 1;3\rangle) \\
s &:= \ (a\ \langle 1;1\rangle, b\ \langle 1;2\rangle, c\ \langle 1;3\rangle) \\
z &:= \ (\epsilon, \epsilon, \epsilon)
\end{aligned}
$$

Here the dimensions are $d(S) = 1$ and $d(N) = 3$ and the arities are $a(c) = a(s) = 1$ and $a(z) = 0$. ϵ is the empty string. This is a simple enough language but at the same time it demonstrates all important aspects of PMCFG. It is also one of the canonical examples of non context-free languages.

The concrete syntax tree for the string $a^n b^n c^n$ is $c\ (s\ (s\ \ldots\ (s\ z)\ldots))$ where s is applied n times. The function z does not have arguments and it corresponds to the base case when $n = 0$. Every application of s over another tree increases n by one. For example the function z is linearized to a tuple with three empty strings but when we apply s twice then we get (aa, bb, cc). Finally the application of c combines all elements in the tuple in a single string i.e. $c\ (s\ (s\ z))$ will produce

the string $aabbcc$.

2.3.1 The Idea

Although PMCFG is not context-free it can be approximated with an overgenerating context-free grammar. The problem with this approach is that the parser produces many spurious parse trees that have to be filtered out. A direct parsing algorithm for PMCFG should avoid this and a careful look at the difference between PMCFG and CFG gives an idea. The context-free approximation of $a^n b^n c^n$ is the language $a^* b^* c^*$ with grammar:

$$
\begin{aligned}
S &\rightarrow ABC \\
A &\rightarrow \epsilon \mid aA \\
B &\rightarrow \epsilon \mid bB \\
C &\rightarrow \epsilon \mid cC
\end{aligned}
$$

The string "$aabbcc$" is in the language and it can be derived with the following steps:

$$
\begin{aligned}
& S \\
\Rightarrow\ & ABC \\
\Rightarrow\ & aABC \\
\Rightarrow\ & aaABC \\
\Rightarrow\ & aaBC \\
\Rightarrow\ & aabBC \\
\Rightarrow\ & aabbBC \\
\Rightarrow\ & aabbC \\
\Rightarrow\ & aabbcC \\
\Rightarrow\ & aabbccC \\
\Rightarrow\ & aabbcc
\end{aligned}
$$

The grammar is only an approximation because there is no enforcement that we will use only equal number of reductions for A, B and C. This can be guaranteed

if we replace B and C with new categories B' and C' after the derivation of A:

$$\begin{aligned} B' &\to bB'' & C' &\to cC'' \\ B'' &\to bB''' & C'' &\to cC''' \\ B''' &\to \epsilon & C''' &\to \epsilon \end{aligned}$$

In this case the only possible derivation from $aaB'C'$ is $aabbcc$.

The parser works like a context-free parser, except that during the parsing it generates fresh categories and rules which are specializations of the originals. The newly generated rules are always versions of already existing rules where some category is replaced with a new more specialized category. The generation of specialized categories prevents the parser from recognizing phrases that are not in the scope of the grammar.

The algorithm is described as a deductive process in the style of Shieber et al. [1995]. The process derives a set of items where each item is a statement about the grammatical status of some substring in the input.

The inference rules are in natural deduction style:

$$\frac{X_1 \ldots X_n}{Y} < \text{side conditions on } X_1, \ldots, X_n >$$

where the premises X_i are some items and Y is the derived item. We assume that $w_1 \ldots w_n$ is the input string.

2.3.2 Deduction Rules

The deduction system deals with three types of items: active, passive and production items.

Productions In Shieber's deduction systems, the grammar is constant and the existence of a given production is specified as a side condition. In our case the grammar is incrementally extended at runtime, so the set of productions is a part of the deduction set. The productions from the original grammar are axioms and are included in the initial deduction set.

Active Items The active items represent the partial parsing result:

$$[^k_j A \to f[\vec{B}]; l : \alpha \bullet \beta], \quad j \leq k$$

The interpretation is that there is a function f with a corresponding production:

$$A \to f[\vec{B}]$$
$$f := (\gamma_1, \dots \gamma_{l-1}, \alpha\beta, \dots \gamma_{r(f)})$$

such that the tree $(f\ t_1 \dots t_{a(f)})$ will produce the substring $w_{j+1} \dots w_k$ as a prefix in constituent l for any sequence of arguments $t_i : B_i$. The sequence α is the part that produced the substring:

$$\mathcal{K}(\mathcal{L}(t_1), \mathcal{L}(t_2) \dots \mathcal{L}(t_{a(f)}))\ \alpha = w_{j+1} \dots w_k$$

and β is the part that is not processed yet.

Passive Items The passive items are of the form:
$[^k_j A; l; N]\ , \quad j \leq k$
and state that there exists at least one production:

$$A \to f[\vec{B}]$$
$$f := (\gamma_1, \gamma_2, \dots \gamma_{r(f)})$$

and a tree $(f\ t_1 \dots t_{a(f)}) : A$ such that the constituent with index l in the linearization of the tree is equal to $w_{j+1} \dots w_k$. Contrary to the active items in the passive the whole constituent is matched:

$$\mathcal{K}(\mathcal{L}(t_1), \mathcal{L}(t_2) \dots \mathcal{L}(t_{a(f)}))\ \gamma_l = w_{j+1} \dots w_k$$

Each time when we complete an active item, a passive item is created and at the same time we create a new category N which accumulates all productions for A that produce the $w_{j+1} \dots w_k$ substring from constituent l. All trees of category N must produce $w_{j+1} \dots w_k$ in the constituent l.

There are six inference rules (see Figure 2.2).

The INITIAL PREDICT rule derives one item spanning the $0 - 0$ range for each production whose result category is mapped to the start category in the abstract syntax.

In the PREDICT rule, for each active item with dot before a $\langle d; r \rangle$ pair and for each production for B_d, a new active item is derived where the dot is in the beginning of constituent r in g.

When the dot is before some terminal s and s is equal to the current terminal w_k then the SCAN rule derives a new item where the dot is moved to the next position.

INITIAL PREDICT

$$\frac{A \to f[\vec{B}]}{[^0_0 A \to f[\vec{B}]; 1 : \bullet\alpha]} \quad \psi_N(A) = S, S \text{ - the start category in } \mathcal{A}, \alpha = \text{rhs}(f, 1)$$

PREDICT

$$\frac{B_d \to g[\vec{C}] \qquad [^k_j A \to f[\vec{B}]; l : \alpha \bullet \langle d; r \rangle\ \beta]}{[^k_k B_d \to g[\vec{C}]; r : \bullet\gamma]} \quad \gamma = \text{rhs}(g, r)$$

SCAN

$$\frac{[^k_j A \to f[\vec{B}]; l : \alpha \bullet s\ \beta]}{[^{k+1}_j A \to f[\vec{B}]; l : \alpha\ s \bullet \beta]} \quad s = w_{k+1}$$

COMPLETE

$$\frac{[^k_j A \to f[\vec{B}]; l : \alpha\bullet]}{N \to f[\vec{B}] \qquad [^k_j A; l; N]} \quad N = (A, l, j, k)$$

COMBINE

$$\frac{[^u_j A \to f[\vec{B}]; l : \alpha \bullet \langle d; r \rangle\ \beta] \qquad [^k_u B_d; r; N]}{[^k_j A \to f[\vec{B}\{d := N\}]; l : \alpha\ \langle d; r \rangle \bullet \beta]}$$

Figure 2.2: Deduction Rules

When the dot is at the end of an active item then it is converted to a passive item in the COMPLETE rule. The category N in the passive item is a fresh category created for each unique (A, l, j, k) quadruple. A new production is derived for N which has the same function and arguments as in the active item.

The item in the premise of COMPLETE was at some point predicted in PREDICT from some other item. The COMBINE rule will later replace the occurence A in the original item (the premise of PREDICT) with the specialization N.

The COMBINE rule has two premises: one active item and one passive. The passive item starts from position u and the only inference rule that can derive items with different start positions is PREDICT. Also the passive item must have been predicted from an active item where the dot is before $\langle d; r\rangle$, the category for argument number d must have been B_d and the item ends at u. The active item in the premise of COMBINE is such an item so it was one of the items used to predict the passive one. This means that we can move the dot after $\langle d; r\rangle$ and the d-th argument is replaced with its specialization N.

If the string β contains another reference to the d-th argument, then the next time when it has to be predicted the rule PREDICT will generate active items, only for those productions that were successfully used to parse the previous constituents. If a context-free approximation was used, this would have been equivalent to unification of the redundant subtrees. Instead this is done at runtime which also reduces the search space.

The parsing is successful if we have derived the $[^n_0 A; 1; A']$ item, where n is the length of the text, $\psi_N(A)$ is equal to the start category and A' is the newly created category.

The parser is incremental because all active items span up to position k and the only way to move to the next position is the SCAN rule where a new symbol from the input is consumed.

2.3.3 A Complete Example

An example sequence of derivation steps for the string abc is shown on Figure 2.3. In the first column we show the derived items and in the second the rule that was applied. The rule name is followed by the line numbers of the items that are premises for the rule.

The first three lines are just the productions from the original grammar. After that we start the real parsing with the rule INITIAL PREDICT. From the item on line 4 we can predict that either function s or z should be applied (lines 5 and 6). The sequence from line 7 to line 15 follows the hypothesis that function z is

1	$S \to c[N]$	
2	$N \to s[N]$	
3	$N \to z[]$	
4	$[^0_0 S \to c[N]; 1 : \bullet\langle 1;1\rangle\langle 1;2\rangle\langle 1;3\rangle]$	Initial Predict 1
5	$[^0_0 N \to s[N]; 1 : \bullet a\langle 1;1\rangle]$	Predict 2 4
6	$[^0_0 N \to z[]; 1 : \bullet]$	Predict 3 4
7	$C_1 \to z[] \quad [^0_0 N; 1; C_1]$	Complete 6
8	$[^0_0 S \to c[C_1]; 1 : \langle 1;1\rangle \bullet \langle 1;2\rangle\langle 1;3\rangle]$	Combine 4 7
9	$[^0_0 C_1 \to z[]; 2 : \bullet]$	Predict 8
10	$C_2 \to z[] \quad [^0_0 C_1; 2; C_2]$	Complete 9
11	$[^0_0 S \to c[C_2]; 1 : \langle 1;1\rangle\langle 1;2\rangle \bullet \langle 1;3\rangle]$	Combine 8 10
12	$[^0_0 C_2 \to z[]; 3 : \bullet]$	Predict 11
13	$C_3 \to z[] \quad [^0_0 C_2; 3; C_3]$	Complete 12
14	$[^0_0 S \to c[C_3]; 1 : \langle 1;1\rangle\langle 1;2\rangle\langle 1;3\rangle\bullet]$	Combine 11 13
15	$C_4 \to c[C_3] \quad [^0_0 S; 1; C_4]$	Complete 14
16	$[^1_0 N \to s[N]; 1 : a \bullet \langle 1;1\rangle]$	Scan 5
17	$[^1_1 N \to s[N]; 1 : \bullet a\langle 1;1\rangle]$	Predict 16
18	$[^1_1 N \to z[]; 1 : \bullet]$	Predict 16
19	$C_5 \to z[] \quad [^1_1 N; 1; C_5]$	Complete 18
20	$[^1_0 N \to s[C_5]; 1 : a\langle 1;1\rangle\bullet]$	Combine 16 19
21	$C_6 \to s[C_5] \quad [^1_0 N; 1; C_6]$	Complete 20
22	$[^1_0 S \to c[C_6]; 1 : \langle 1;1\rangle \bullet \langle 1;2\rangle\langle 1;3\rangle]$	Combine 4 21
23	$[^1_1 C_6 \to s[C_5]; 2 : \bullet b\langle 1;2\rangle]$	Predict 22
24	$[^2_1 C_6 \to s[C_5]; 2 : b \bullet \langle 1;2\rangle]$	Scan 23
25	$[^2_2 C_5 \to z[]; 2 : \bullet]$	Predict 24
26	$C_7 \to z[] \quad [^2_2 C_5; 2; C_7]$	Complete 25
27	$[^2_1 C_6 \to s[C_7]; 2 : b\langle 1;2\rangle\bullet]$	Combine 24 26
28	$C_8 \to s[C_7] \quad [^2_1 C_6; 2; C_8]$	Complete 27
29	$[^2_0 S \to c[C_8]; 1 : \langle 1;1\rangle\langle 1;2\rangle \bullet \langle 1;3\rangle]$	Combine 22 28
30	$[^2_2 C_8 \to s[C_7]; 3 : \bullet c\langle 1;3\rangle]$	Predict 29
31	$[^3_2 C_8 \to s[C_7]; 3 : c \bullet \langle 1;3\rangle]$	Scan 30
32	$[^3_3 C_7 \to z[]; 3 : \bullet]$	Predict 31
33	$C_9 \to z[] \quad [^3_3 C_7; 3; C_9]$	Complete 32
34	$[^3_2 C_8 \to s[C_9]; 3 : c\langle 1;3\rangle\bullet]$	Combine 31 33
35	$C_{10} \to s[C_9] \quad [^3_2 C_8; 3; C_{10}]$	Complete 34
36	$[^3_0 S \to c[C_{10}]; 1 : \langle 1;1\rangle\langle 1;2\rangle\langle 1;3\rangle\bullet]$	Combine 29 35
37	$C_{11} \to c[C_{10}] \quad [^3_0 S; 1; C_{11}]$	Complete 36

Figure 2.3: The deduction sequence for parsing the string abc

applied. At the end we deduce the passive item $[^0_0 S; 1; C_4]$ which is for the start category but does not span the whole sentence so we cannot use this item as a final item. The deduction follows with lines 16-22 which rely on the hypothesis that the tree should start with function s (this was predicted on line 5). In this derivation fragment we have fully recognized the symbol a and the dot is again in front of the argument $\langle 1; 1 \rangle$ (line 16). At this point we can again predict that the next function is either z or s. However, if the next function was s, then the next symbol must be a which is not the case so we cannot continue with this hypothesis (line 17). If we continue with function z, then we can just complete with the empty string and move the dot in item 16 after argument $\langle 1; 1 \rangle$ which completes this item as well (lines 18-21). Having done this we can also move the dot on line 4 which produces the item on line 22. Note that now the argument to function c is changed from N to C_6. We have done similar replacements all the way but this is the first point where this really leads to some restrictions. We have created two new productions:

$$
\begin{aligned}
C_6 &\rightarrow s[C_5] \\
C_5 &\rightarrow z[]
\end{aligned}
$$

which say that the only concrete syntax tree that we can construct for category C_6 is $s\ z$. This encodes the fact that we have recognized only one token a. When we continue with the recognition of the next token b then we will do prediction with category C_6 instead of the original N (lines 23-29). Since $s\ z$ is the only allowed expression exactly one b will be allowed. After its recognition a new category C_8 will be created along with the productions:

$$
\begin{aligned}
C_8 &\rightarrow s[C_7] \\
C_7 &\rightarrow z[]
\end{aligned}
$$

This set of productions is the same as the one for categories C_6 and C_5 but this time we encode the fact we have recognized both the tokens a and b. Since the recognition of b does not place any further constraint on the possible analyses we get isomorphic sets of productions. Finally we recognize the last token c by doing predictions from category C_8 (lines 30-36). The last item 37 just completes the item on line 36. The result is a passive item for category S spanning over the whole sentence so we have successfully recognized the input.

Note that up to the point where we have recognized the first part of the sentence i.e. the token a we basically do context-free parsing with just a little bit of extra

bookkeeping. After that point, however, we use the new more restricted categories and at this point the parsing becomes deterministic. We do not search for a parse tree anymore but just check that the rest of the sentence is consistent with what we expect. This shows how the parsing with PMCFG can in some cases be more efficient than parsing with approximating CFG followed by postprocessing.

2.3.4 Soundness

The parsing system is sound if every derivable item represents a valid grammatical statement under the interpretation given to every type of item.

The derivation in INITIAL PREDICT and PREDICT is sound because the item is derived from an existing production and the string before the dot is empty so:

$$\mathcal{K}\ \sigma\ \epsilon = \epsilon$$

The rationale for SCAN is that if

$$\mathcal{K}\ \sigma\ \alpha = w_{j-1} \ldots w_k$$

and $s = w_{k+1}$ then

$$\mathcal{K}\ \sigma\ (\alpha\ s) = w_{j-1} \ldots w_{k+1}$$

If the item in the premise is valid, then it is based on an existing production and function and so will be the item in the consequent.

In the COMPLETE rule, the dot is at the end of the string. This means that $w_{j+1} \ldots w_k$ will be not just a prefix in constituent l of the linearization but the full string. This is exactly what is required in the semantics of the passive item. The passive item is derived from a valid active item so there is at least one production for A. The category N is unique for each (A, l, j, k) quadruple so it uniquely identifies the passive item in which it is placed. There might be many productions that can produce the passive item but all of them should be able to generate $w_{j+1} \ldots w_k$ and they are exactly the productions that are added to N. From all these arguments it follows that COMPLETE is sound.

The COMBINE rule is sound because from the active item in the premise we know that:

$$\mathcal{K}\ \sigma\ \alpha = w_{j+1} \ldots w_u$$

for every context σ built from the trees:

$$t_1 : B_1;\ t_2 : B_2;\ \ldots t_{a(f)} : B_{a(f)}$$

From the passive item we know that every production for N produces the $w_{u+1} \ldots w_k$ in r. From that follows that

$$\mathcal{K}\ \sigma'\ (\alpha\langle d; r\rangle) = w_{j+1} \ldots w_k$$

where σ' is the same as σ except that B_d is replaced with N. Note that the last conclusion will not hold if we were using the original context because B_d is a more general category and can contain productions that do not derive $w_{u+1} \ldots w_k$.

2.3.5 Completeness

The parsing system is complete if it derives an item for every valid grammatical statement. In our case we have to prove that for every possible parse tree the corresponding items will be derived.

The proof for completeness requires the following lemma:

Lemma 1 *For every possible concrete syntax tree*

$$(f\ t_1 \ldots t_{a(f)}) : A$$

with linearization

$$\mathcal{L}(f\ t_1 \ldots t_{a(f)}) = (x_1, x_2 \ldots x_{d(A)})$$

where $x_l = w_{j+1} \ldots w_k$*, the system will derive an item* $[^k_j A; l; A']$ *if the item* $[^j_j A \rightarrow f[\vec{B}]; l : \bullet\alpha_l]$ *was predicted before that. We assume that the function definition is:*

$$f := (\alpha_1, \alpha_2 \ldots \alpha_{r(f)})$$

The proof is by induction on the depth of the tree. If the tree has only one level, then the function f does not have arguments and from the linearization definition and from the premise in the lemma it follows that $\alpha_l = w_{j+1} \ldots w_k$. From the active item in the lemma by applying iteratively the SCAN rule and finally the COMPLETE rule the system will derive the requested item.

If the tree has subtrees, then we assume that the lemma is true for every subtree and we prove it for the whole tree. We know that

$$\mathcal{K}\ \sigma\ \alpha_l = w_{j+1} \dots w_k$$

Since the function $\mathcal{K}$ does simple substitution it is possible for each $\langle d; s \rangle$ pair in α_l to find a new range in the input string $j' - k'$ such that the lemma to be applicable for the corresponding subtree $t_d : B_d$. The terminals in α_l will be processed by the SCAN rule. Rule PREDICT will generate the active items required for the subtrees and the COMBINE rule will consume the produced passive items. Finally the COMPLETE rule will derive the requested item for the whole tree.

From the lemma we can prove the completeness of the parsing system. For every possible tree $t : S$ such that $\mathcal{L}(t) = (w_1 \dots w_n)$ we have to prove that the $[^n_0 S; 1; S']$ item will be derived. Since the top-level function of the tree must be from production for S the INITIAL PREDICT rule will generate the active item in the premise of the lemma. From this and from the assumptions for t, it follows that the requested passive item will be derived.

2.3.6 Complexity

The algorithm is very similar to the Earley [1970] algorithm for context-free grammars. The similarity is even more apparent when the inference rules in this section are compared to the inference rules for the Earley algorithm presented in Shieber et al. [1995] and Ljunglöf [2004]. This suggests that the space and time complexity of the PMCFG parser should be similar to the complexity of the Earley parser which is $\mathcal{O}(n^2)$ for space and $\mathcal{O}(n^3)$ for time. However we generate new categories and productions at runtime and this has to be taken into account.

Let the $\mathcal{P}(j)$ function be the maximal number of productions generated from the beginning up to the state where the parser has just consumed terminal number j. $\mathcal{P}(j)$ is also the upper limit for the number of categories created because in the worst case there will be only one production for each new category.

The active items have two variables that directly depend on the input size - the start index j and the end index k. If an item starts at position j, then there are $(n - j + 1)$ possible values for k because $j \leq k \leq n$. The item also contains a production and there are $\mathcal{P}(j)$ possible choices for it. In total there are:

$$\sum_{j=0}^{n} (n - j + 1)\mathcal{P}(j)$$

possible choices for one active item. The possibilities for all other variables are only a constant factor. The $\mathcal{P}(j)$ function is monotonic because the algorithm only adds new productions and never removes. From that follows the inequality:

$$\sum_{j=0}^{n}(n-j+1)\mathcal{P}(j) \leq \mathcal{P}(n)\sum_{i=0}^{n}(n-j+1)$$

which gives the approximation for the upper limit:

$$\mathcal{P}(n)\frac{n(n+1)}{2}$$

The same result applies to the passive items. The only difference is that the passive items have only a category instead of a full production. However the upper limit for the number of categories is the same. Finally the upper limit for the total number of active, passive and production items is:

$$\mathcal{P}(n)(n^2+n+1)$$

The expression for $\mathcal{P}(n)$ is grammar dependent but we can estimate that it is polynomial because the set of productions corresponds to the compact representation of all parse trees in the context-free approximation of the grammar. The exponent however is grammar dependent. From this we can expect that asymptotic space complexity will be $\mathcal{O}(n^e)$ where e is some parameter for the grammar. This is consistent with the results in Nakanishi et al. [1997] and Ljunglöf [2004] where the exponent also depends on the grammar.

The time complexity is proportional to the number of items and the time needed to derive one item. The time is dominated by the most complex rule which in this algorithm is COMBINE. All variables that depend on the input size are present both in the premises and in the consequent except u. There are n possible values for u so the time complexity is $\mathcal{O}(n^{e+1})$.

2.3.7 Tree Extraction

If the parsing is successful, then we need a way to extract the syntax trees. Everything that we need is already in the set of newly generated productions. If the start category is S, then we look up all passive items of the form $[^n_0 A; 0; A']$ where $\psi_N(A) = S$ and A' is a newly produced concrete category. Every tree t of category A' is a concrete syntax tree for the input sentence (see Definition 5, Section 2.1).

In the example on Figure 2.3 the goal item is $[^3_0 S; 1; C_{11}]$ and if we follow the newly generated category C_{11}, then the set of all accessible productions is:

$$\begin{array}{ll} C_{11} & \rightarrow c[C_{10}] \\ C_{10} & \rightarrow s[C_9] \\ C_9 & \rightarrow z[] \end{array}$$

With this set, the only concrete syntax tree that can be constructed is $c\ (s\ z)$ and this is the only tree that can produce the string abc. From the concrete syntax tree we can obtain the abstract syntax tree by mapping every function symbol to its abstract counterpart. Formally we can define the mapping ψ_{tr} which turns every concrete syntax tree into an abstract syntax tree:

$$\psi_{tr}(f\ t_1 \ldots t_{a(f)}) = \psi_F(f)\ \psi_{tr}(t_1) \ldots \psi_{tr}(t_{a(f)})$$

Since the mapping $\psi_F(f)$ is a many to one relation the same applies to ψ_{tr}. This has to be taken into account because the parser can find several concrete syntax trees that are all mapped to the same abstract tree. The tree extraction procedure should simply eliminate the duplicated trees.

Note that the grammar can be erasing; i.e., there might be functions which ignore some of their arguments, for example:

$$\begin{array}{l} S \rightarrow f[B_1, B_2, B_3] \\ f := (\langle 1; 1\rangle \langle 3; 1\rangle) \end{array}$$

There are three arguments but only two of them are used. When a sentence is parsed this will generate a new specialized production:

$$S' \rightarrow f[B'_1, B_2, B'_3]$$

Here S,B_1 and B_3 are specialized to S', B'_1 and B'_3 but the B_2 category is still the same. This is correct because any subtree for the second argument will produce the same sentence. This is actually a very common situation when we have dependent types in the abstract syntax, since often some of the dependencies are not linearized in the concrete syntax. In this case, despite that for the parser any value for the second argument is just as good, there is only one value which is consistent with the semantic restrictions. When doing tree extraction, such erased arguments are replaced with metavariables, i.e. we get:

$$f\ t_1\ ?0\ t_2$$

where t_1 and t_2 are some subtrees and $?0$ is the metavariable. The metavariables might be later substituted by the type checker or might remain intact in which case they are interpreted as placeholders which can hold any value. The tree extractor knows when to produce metavariables because the category for the unused arguments is in the original set of categories $N^{\mathcal{C}}$ in the grammar.

Just like with the context-free grammars the parsing algorithm is polynomial but the chart can contain an exponential or even infinite number of trees. Despite this the chart is a compact finite representation of the set of trees.

2.3.8 Implementation

Every implementation requires a careful design of the data structures. For efficient access the set of items is split into four subsets: $\mathbb{A}$, $\mathbb{S}_j$, $\mathbb{C}$ and $\mathbb{P}$. $\mathbb{A}$ is the agenda, i.e. the set of active items that have to be analyzed. $\mathbb{S}_j$ contains items for which the dot is before an argument reference and which span up to position j. For fast lookup the set $\mathbb{S}_j$ is organized as a multimap where the key is a pair of a concrete category A and a constituent index r and the value is an item such that the argument reference points to constituent r of category A. $\mathbb{C}$ is the set of possible continuations, i.e. a set of items for which the dot is just after a terminal. Just like $\mathbb{S}_j$, $\mathbb{C}$ is a map indexed by the value of the terminal. $\mathbb{P}$ is the set of productions. In addition, the set $\mathbb{F}$ is used internally for the generation of fresh categories. It is a map $(j, A, r) \mapsto N$ from the start position j, the concrete category A and the constituent index r to the newly generated category N. Every time when we have to generate a new category, we lookup in $\mathbb{F}$ and if there is already a category created for the triple (j, A, r) then we reuse it. Otherwise we create a new category.

The pseudocode of the implementation is given in Figure 2.4. There are two procedures *Init* and *Compute*.

Init computes the initial values of $\mathbb{S}$, $\mathbb{P}$ and $\mathbb{A}$. The initial agenda $\mathbb{A}$ is the set of all items that can be predicted from any concrete category $S^{\mathcal{C}}$ which maps to the abstract start category S (INITIAL PREDICT rule).

Compute consumes items from the current agenda and applies the SCAN, PREDICT, COMBINE or COMPLETE rules. The case statement matches the current item against the patterns of the rules and selects the proper rule. The PREDICT and COMBINE rules have two premises so they are used in two places. In both cases one of the premises is related to the current item and a loop is needed to find item matching the other premis.

The passive items are not independent entities but are just the combination of

```
procedure Init() {
  k = 0
  𝕊_i = ∅,   for every i
  ℙ = the set of productions P in the grammar

  𝔸 = ∅
  forall S^C → f[B⃗] ∈ P where ψ_N(S^C) = S do   // INITIAL PREDICT
    𝔸 = 𝔸 + [0; S^C → f[B⃗]; 1; 0]

  return (𝕊, ℙ, 𝔸)
}


procedure Compute(k, (𝕊, ℙ, 𝔸)) {
  ℂ = ∅
  𝔽 = ∅
  while 𝔸 ≠ ∅ do {
    let x ∈ 𝔸,   x ≡ [j; A → f[B⃗]; l; p]
    𝔸 = 𝔸 − x
    case the dot in x is {
      before s ∈ T ⇒ ℂ = ℂ + (s ↦ [j; A → f[B⃗]; l; p + 1])   // SCAN

      before ⟨d; r⟩  ⇒ if ((B_d, r) ↦ (x, d)) ∉ 𝕊_k then {
                           𝕊_k = 𝕊_k + ((B_d, r) ↦ (x, d))
                           forall B_d → g[C⃗] ∈ ℙ do   // PREDICT
                             𝔸 = 𝔸 + [k; B_d → g[C⃗]; r; 0]
                         }
                         forall (k; B_d, r) ↦ N ∈ 𝔽 do   // COMBINE
                           𝔸 = 𝔸 + [j; A → f[B⃗{d := N}]; l; p + 1]

      at the end     ⇒ if ∃N.((j, A, l) ↦ N ∈ 𝔽) then {
                           forall (N, r) ↦ (x′, d′) ∈ 𝕊_k do     // PREDICT
                             𝔸 = 𝔸 + [k; N → f[B⃗]; r; 0]
                         } else {
                           generate fresh N     // COMPLETE
                           𝔽 = 𝔽 + ((j, A, l) ↦ N)
                           forall (A, l) ↦ ([j′; A′ → f′[B⃗′]; l′; p′], d) ∈ 𝕊_j do   // COMBINE
                             𝔸 = 𝔸 + [j′; A′ → f′[B⃗′{d := N}]; l′; p′ + 1]
                         }
                         ℙ = ℙ + (N → f[B⃗])
    }
  }

  return (𝕊, ℙ, ℂ)
}
```

Figure 2.4: Pseudocode of the parser implementation

key and value in the set $\mathbb{F}$. Only the start position of every item is kept because the end position for the interesting passive items is always the current position and the active items are either in the agenda if they end at the current position or they are in the $\mathbb{S}_j$ set if they end at position j. The active items also keep only the dot position in the constituent because the constituent definition can be retrieved from the grammar. For this reason the runtime representation of the items is $[j; A \to f[\vec{B}]; l; p]$ where j is the start position of the item and p is the dot position inside the constituent.

The *Compute* function returns the updated $\mathbb{S}$ and $\mathbb{P}$ sets and the set of possible continuations $\mathbb{C}$. The set of continuations is a map indexed by a terminal and the values are active items. The parser computes the set of continuations at each step and if the current terminal is one of the keys the set of values for it is taken as an agenda for the next step.

In the concrete Haskell implementation, the core parsing API consists of four functions: *initState*, *nextState*, *getCompletions* and *getParseOutput*. The initState function:

$$\mathit{initState} :: \mathit{PGF} \to \mathit{Language} \to \mathit{Type} \to \mathit{ParseState}$$

corresponds to function *Init* in Figure 2.4. It takes a grammar in *PGF* format, a *Language* and a start category, given more generally as a *Type*, and returns the initial *ParseState* which is a Haskell structure encapsulating the sets $\mathbb{S}$, $\mathbb{P}$ and $\mathbb{A}$.

The input sentence is consumed by feeding the tokens one by one to function *nextState*:

$$\mathit{nextState} :: \mathit{ParseState} \to \mathit{ParseInput} \to \mathit{Either}\ \mathit{ErrorState}\ \mathit{ParseState}$$

It takes as arguments the last parse state and a new token, encapsulated as a *ParseInput* structure, and either returns a new parse state or an error state. If an error state is returned then this indicates that the sentence is not in the coverage of the grammar. The information in the error state is used for different error recovery strategies. The new parse state contains the updated $\mathbb{S}$ and $\mathbb{P}$ sets and a new agenda $\mathbb{A}$ which is retrieved from the continuation $\mathbb{C}$, computed by function *Compute*.

When all tokens are consumed, the final result can be obtained by applying the function *getParseOutput* to the final parse state:

$$\begin{aligned}\mathit{getParseOutput} &:: \mathit{ParseState} \to \mathit{Type} \to \mathit{Maybe}\ \mathit{Int} \\ &\to (\mathit{ParseOutput}, \mathit{BracketedString})\end{aligned}$$

The result is a pair of *ParseOutput* and *BracketedString*, where *ParseOutput* represents the list of abstract syntax trees or an error message if the parsing is not successful, and *BracketedString* represents the parse tree for the sentence. The function also takes as additional arguments the start category as a *Type* and an eventual depth limitation if a proof search is necessary during the tree extraction (this will be clarified in Section 3.5).

From every *ParseState*, it is also possible to compute the set of possible next tokens. The function *getCompletions*:

$$getCompletions :: ParseState \rightarrow String \rightarrow Map\ Token\ ParseState$$

calls *Compute* and transforms the continuations set $\mathbb{C}$ into a map from token to parse state. The extra string argument to the function is a filter which limits the set of tokens to only those that start with the same prefix.

2.3.9 Evaluation

The algorithm was evaluated with all languages from the GF Resource Grammar Library [Ranta, 2009]. Although these grammars are not designed for parsing they are still a good parsing benchmark because these are the biggest GF grammars. The sizes of the compiled grammars in terms of number of productions and number of unique discontinuous constituents can be seen on Table 2.1 in Section 2.8.1. The number of constituents roughly corresponds to the number of productions in the context-free approximation of the grammar.

In the evaluation 34272 sentences are parsed and the average time for processing a given number of tokens is measured. Figure 2.5 shows the time in milliseconds needed to parse a sentence of a given length. As it can be seen, although the theoretical complexity is polynomial, the real-time performance for practically interesting grammars tends to be linear. The average time per token in milliseconds is shown on Figure 2.6. The slowest grammar is for Finnish, after that are German and Italian, followed by the Romance languages French, Spanish and Catalan plus Bulgarian and Interlingua. The most efficient is the English grammar followed by the Scandinavian languages Danish, Norwegian and Swedish.

2.4 Online Parsing

We pointed out that one of the distinguishing features of our parsing algorithm is that it is incremental. The incrementality, however, is still hidden because despite

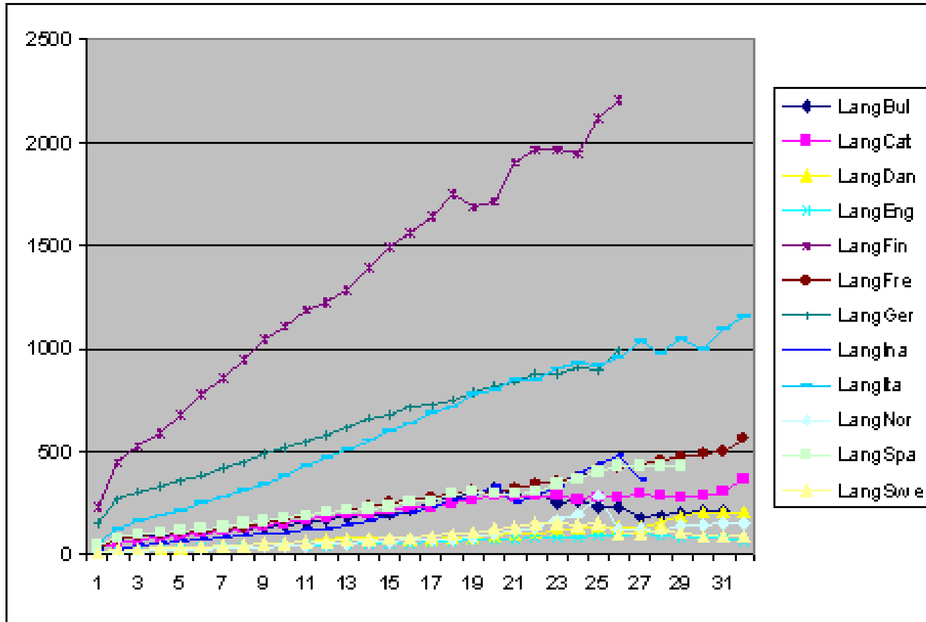


Figure 2.5: Time in milliseconds to parse a sentence of given length

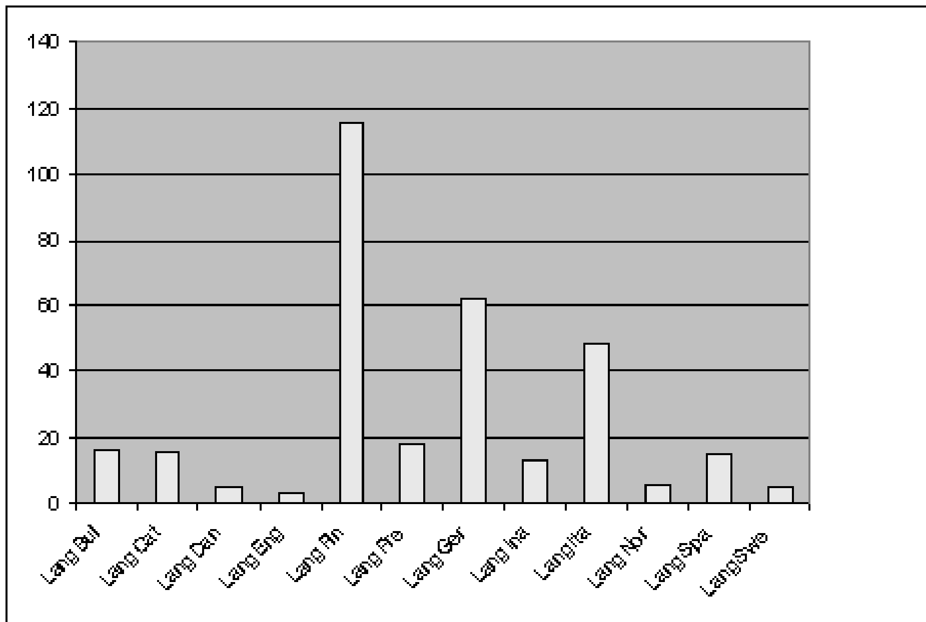


Figure 2.6: Parser performance in milliseconds per token

that the chart contains only the trees that are consistent with the consumed input, the parser does not immediately produce a syntactic analysis of the string. Often the term online parser is used for a parser which is able to produce an analysis before the complete input is consumed. Fortunately, with a modification of the tree extraction procedure, our incremental parser can be turned into an online parser. The analysis that we produce is the original input string where every completely recognized phrase is annotated with its abstract category and its abstract syntax tree. In the Haskell implementation this annotated string is represented with the *BracketedString* structure that is returned from the *getParseOutput* function (see Section 2.3.8 for the signature).

For instance, if we have the grammar:

$$\begin{array}{l} \textbf{fun}\ \mathit{barcelona} : \mathit{Location}; \\ \ \textbf{lin}\ \mathit{barcelona} = \{s = \text{"Barcelona"}\}; \\ \\ \textbf{fun}\ \mathit{welcome} : \mathit{Location} \rightarrow \mathit{Name} \rightarrow \mathit{S}; \\ \ \textbf{lin}\ \mathit{welcome}\ l\ n = \text{"Welcome to"} +\!\!+\ l.s\ +\!\!+\ \text{","}\ +\!\!+\ n.s; \end{array}$$

then the prefix "Welcome to Barcelona ," should be annotated as:

$$\text{"Welcome" "to"}\ (\mathit{Location{:}1{:}2}\ \text{"Barcelona"})\ \text{","}$$

$$?2 = \mathit{barcelona}$$

Here "Barcelona" is the only complete phrase so it is annotated with its category, i.e. *Location*, and with the right constituent index. In this case, *Location* has only one constituent so 1 corresponds to the field s in the grammar. In addition, every phrase is marked with a metavariable which links the phrases with their abstract syntax. In the example, we used the metavariable $?2$ which is bound to the abstract constant *barcelona*.

Since the concrete syntax trees in the parse chart are not directly related to the input string, the way to compute the annotations is to linearize the trees back to strings where during the linearization we put annotations in the right places. The problem is that when we have incomplete sentence, then there is no single tree whose linearization will generate the whole sequence of tokens. Instead the incomplete sentence is a mix of fragments for which there are complete trees and fragments which consist of still uninterpreted tokens. In the example, the

fragments "Welcome to" and "," are just tokens while "Barcelona" should be annotated as a phrase of category *Location*.

The problem can be solved if we start the linearization not from a concrete syntax tree but from a pair $(\vec{B}, \alpha)$ where $\vec{B}$ is a vector of concrete categories and α is a sequence of tokens and $\langle d; r \rangle$ pairs referring to the components of $\vec{B}$. In this representation, the fragments for which there is a complete interpretation will be represented with $\langle d; r \rangle$ where B_d will be the category of the tree in the parse chart, and the uninterpreted tokens will be just kept as tokens.

In the computation of the representation $(\vec{B}, \alpha)$, we need a new kind of items which are similar to the active items in the parsing algorithm but with some irrelevant parts stripped out. For every active item $[^k_j; A \to f[\vec{B}]; l : \alpha \bullet \beta]$ where k is the last position in the incomplete sentence and j is the start position of the item, we define an interpretation item $[j; A; \vec{B}; l : \alpha]$ which contains only the parts that are relevant to the current input. For instance β is not included because it is only something that we expect but have not seen yet. At the same time, the erasure of the function symbol f lets us concatenate several active items into a single interpretation item. For instance, if we have one active item for the range $i - j$ and another for $j - k$, then we can combine them into a single interpretation item which spans from position i to the last position k. Since the different active items can have different function symbols we just removed the function from the interpretation item.

We start the computation with an initial set of interpretation items derived from the last continuation set $\mathbb{C}$ (returned from the procedure *Compute* on Figure 2.4). In this set, every item with a start position j represents a possible interpretation of the range from position j to the end of the consumed input. Now the goal is to extend the scope of every item until we reach the initial position 0 which means that we have recovered the complete interpretations. Since every active item, except the initial one, was predicted from some other item, we can extend the interpretations by looking in the chart for matching active items. Formally this is done by applying the rule:

$$\frac{[j; A; \vec{B}; l : \alpha]}{[i; C; \vec{D}\vec{B}; r : \gamma(\alpha \uparrow a(h))]} \qquad (A, l) \mapsto ([^j_i C \to h[\vec{D}]; r : \gamma \bullet \delta], d) \in \mathbb{S}_j, \quad i < j$$

Here, if we have an interpretation item $[j; A; \vec{B}; l : \alpha]$, then we look for an active item that was suspended at position j and is expecting a phrase of category A for a constituent with index l. This is precisely the set of active items that were the premises of the PREDICT rule. The members of the set are easy to find by looking

in the map $\mathbb{S}_j$ for items indexed by the key (A, l). For every retrieved active item, we generate a new extended interpretation item by concatenating the sequences γ and α and the vectors $\vec{D}$ and $\vec{B}$. Furthermore, since the sequence α has references to the vector $\vec{B}$ and we has appended the vector to the end of $\vec{D}$, we have to update the pairs $\langle d; r\rangle$ in α by incrementing the indices d with the size of $\vec{D}$, i.e. with the arity $a(h)$. In the rule, $\alpha \uparrow n$ denotes a sequences derived from α by increasing the argument indices with n. Finally, we require that $i < j$ which ensures that we do not concatenate empty sequences and guarantees that the algorithm will not go into an infinite loop. In the final set, we will have only items like $[0; A; \vec{B}; l : \alpha]$ which span over the whole input.

Once we have the set of final interpretation items, the pair $(\vec{B}, \alpha)$ from every item corresponds to one possible analysis of the consumed input. The pairs are linearized into annotated (bracketed) strings in the same way as we defined in Section 2.1, except that when we have a pair $\langle d; r\rangle$, then we also put an annotation for $\psi_N(B_d)$ with constituent index r.

Note that since α is composed of already recognized sequences from consecutive active items, the linearization of every possible pair $(\vec{B}, \alpha)$ is exactly equal to the consumed input and they only differ in the annotations. A typical case is the PP-attachement problem. If we have the sentence "Welcome to Barcelona in the summer", then it has two possible annotations:

$($*VP* "Welcome" "to" $($*NP* $($*NP* "Barcelona"$)$ $($*PP* "in" "the" "summer"$)))$
$($*VP* $($*VP* "Welcome" "to" $($*NP* "Barcelona"$))$ $($*PP* "in" "the" "summer"$))$

This situation is in general unavoidable and here we have two choices. Either we rank the set of analyses and pick the one with the highest rank or we compute an analysis which is the intersection of all possible analyses. In the example, we should keep only those annotations that exists in all possible analyses:

$($*VP* "Welcome" "to" $($*NP* "Barcelona"$)$ $($*PP* "in" "the" "summer"$))$

It is also possible to combine the two scenarios by computing the intersection of the top N analyses with the highest rank. Here we will concentrate only on the intersection since the ranking is at least in principle a trivial operation.

Since an annotation is put only when we linearize an argument reference $\langle d; r\rangle$ and since the corresponding category B_d corresponds to a node in the forest of concrete syntax trees, the way to get the intersection is to add the restriction that we put an annotation only if the node B_d is shared among all trees. More concretely, a node labelled with the category A in the forest of concrete syntax trees

is characterized by a set of productions:

$$A \to f_1[\vec{B_1}], \quad A \to f_2[\vec{B_2}], \quad \ldots \quad A \to f_n[\vec{B_n}]$$

where every production has the same category on the left-hand side but different functions and arguments. Since every argument denotes another node in the forest, then starting from any node, we can extract a list of trees by recursion over the chain of productions. Typically the extracted trees will share many subtrees because they are built by going through common nodes in the forest. We can detect the sharing by computing for every node A the set of nodes $t(A)$ shared between all trees rooted at A. The equation for $t(A)$ is defined recursively:

$$t(A) = \{A\} \cup \bigcap_{i=1}^{n} \bigcup_{j=1}^{a(f_i)} t(B_{ij})$$

In the trivial case, when none of the productions for A has arguments, $t(A)$ is a singleton set containing just A. Obviously the node A is always shared between all the trees. When the productions have arguments, then for every production we compute the union of the sets $t(B_{ij})$ where B_{ij} is the j-th argument of the i-th production. Finally we add the intersection of the unions to the singleton set containing A.

The equation for $t(A)$ is trivially computable, if the forest is acyclic. In the presence of cycles, however, the value for $t(A)$ will directly or indirectly depend on itself, so in general the above is a fixed point equation which can be computed by using the Knaster-Tarski theorem [Tarski, 1955]. More concretely, we are interested in the greatest fixed point since we want to preserve as many annotations as possible.

Taking into account the idempotence property of the union and the intersection of sets any equation which relates $t(A)$ to itself can be reduced to:

$$t(A) = (t(A) \cup L) \cap M$$

where L and M are some sets of nodes. The Knaster-Tarski theorem tells us that we can get the greatest fixed point by starting from the universal set and by applying the equation iteratively. In our case, by substituting $t(A)$ in the right-hand side of the equation with the universal set, we get to the fixed point $t(A) = M$ immediately. In the implementation, we rely on the observation that a single iteration is enough, and we compute $t(A)$ as if the forest was acyclic. If we reach a cycle, then it is resolved by substituting $t(A)$ with the universal set.

2.5 Linearization

The linearization of abstract syntax trees to natural language is a much easier process than parsing. In Section 2.1 we already defined how a string or a tuple of strings is derived from a concrete syntax tree. The concrete syntax tree, however, is an internal concept in the framework so we want to have direct linearization of the abstract trees. For this we must find all concrete trees that map to a given abstract tree.

The search is more efficient if we decompose the set of productions P in the concrete syntax into a family of non-intersecting subsets $P_{f^{\mathcal{A}},\vec{B}} \subset P$ where every subset contains only productions whose arguments categories are $\vec{B}$ and whose concrete functions map to the same abstract function $f^{\mathcal{A}}$:

$$P_{f^{\mathcal{A}},\vec{B}} = \{A \to f^{\mathcal{C}}[\vec{B}] \mid A \to f^{\mathcal{C}}[\vec{B}] \in P;\ \psi_F(f^{\mathcal{C}}) = f^{\mathcal{A}}\}$$

The decomposition is computable because the set P is finite so we simply group the different productions into families with different functions and categories.

We are now ready to define the transformation $t^{\mathcal{A}} \Downarrow t^{\mathcal{C}} : B$ from an abstract tree $t^{\mathcal{A}}$ to a concrete tree $t^{\mathcal{C}}$ of category B. Since the transformation is nondeterministic it is best described as a deduction rule:

Abstract to Concrete

$$\frac{t_1^{\mathcal{A}} \Downarrow t_1^{\mathcal{C}} : B_1 \quad \dots \quad t_n^{\mathcal{A}} \Downarrow t_n^{\mathcal{C}} : B_n}{f^{\mathcal{A}}\ t_1^{\mathcal{A}} \dots t_n^{\mathcal{A}} \Downarrow f^{\mathcal{C}}\ t_1^{\mathcal{C}} \dots t_n^{\mathcal{C}} : A} \quad A \to f^{\mathcal{C}}[\vec{B}] \in P_{f^{\mathcal{A}},\vec{B}} \quad n = a(f^{\mathcal{A}}) = a(f^{\mathcal{C}})$$

The rule is bottom-up[3], i.e. if we have to transform a tree where some function is applied to a list of arguments, then we first transform the arguments. The choice of the concrete function in the final tree is nondeterministic because every production in the set $P_{f^{\mathcal{A}},\vec{B}}$ is a possible candidate. Here we get the vector $\vec{B}$ from the linearization of the arguments and $f^{\mathcal{A}}$ is the abstract function in the input tree.

Once we have the concrete syntax tree we can linearize it in the way described in Section 2.1. This gives us a two step process for linearizing abstract syntax trees. First we transform $t^{\mathcal{A}}$ into a concrete syntax tree $t^{\mathcal{A}} \Downarrow t^{\mathcal{C}} : B$ and after that we compute the linearization $\mathcal{L}(t^{\mathcal{C}})$. Fortunately the two steps can be done at once. We define the direct linearization of the abstract tree $t^{\mathcal{A}}$ to a tuple of strings σ in concrete category B as the transformation $t^{\mathcal{A}} \Downarrow \sigma : B$, where the dimension

[3]The linearization can be done in both bottom-up and top-down fashion. Many thanks to Lauri Alanko who pointed out that the bottom-up approach is more efficient.

of σ must be $d(B)$. The deduction rule for direct linearization is:

LINEARIZE

$$\frac{t_1^{\mathcal{A}} \Downarrow \sigma_1 : B_1 \quad \ldots \quad t_n^{\mathcal{A}} \Downarrow \sigma_n : B_n}{f^{\mathcal{A}}\ t_1^{\mathcal{A}} \ldots t_n^{\mathcal{A}} \Downarrow (\mathcal{K}\ \vec{\sigma}\ \alpha_1, \ldots \mathcal{K}\ \vec{\sigma}\ \alpha_{d(A)}) : A}\ A \rightarrow f^{\mathcal{C}}[\vec{B}] \in P_{f^{\mathcal{A}},\vec{B}},\ \ \alpha_i = \text{rhs}(f^{\mathcal{C}}, i)$$

The linearization rule is obtained from the previous rule by substituting in the premises the concrete trees with their linearizations $\sigma_i = \mathcal{L}(t_i^{\mathcal{C}})$ and by substituting in the conclusion the final concrete tree with its linearization:

$$\mathcal{L}(f^{\mathcal{C}}\ t_1^{\mathcal{C}} \ldots t_n^{\mathcal{C}}) = (\mathcal{K}\ \vec{\sigma}\ \alpha_1, \ldots \mathcal{K}\ \vec{\sigma}\ \alpha_{d(A)})$$

The rule LINEARIZE covers the most common case but it cannot be applied, if the abstract tree is not complete. In the discussion for the tree extraction, we mentioned that when the grammar is erasing, the parser will produce incomplete abstract syntax trees where the holes are filled in with metavariables. Linearization of incomplete trees is also allowed which means that we need a linearization rule for metavariables. GF provides a mechanism to define default linearizations for every category. The judgement:

$$\textbf{lindef}\ C\ x = e$$

defines that the expression e is the default linearization for category C. Here x is an argument of type *Str* which can be any string and is used in e for computing the linearization. The linearization of a metavariable is just the application of the default linearization over the name of the variable.

Just like with the ordinary linearization rules, the rules for default linearization are compiled into one or more PMCFG functions. The difference, however, is that for the default linearizations we do not produce PMCFG productions, instead we just remember the set of available functions for every category. We denote the set of default linearization functions for a given concrete category A as $\textit{lindef}(A)$. Now the rule for linearization of metavariables is:

LINEARIZE META

$$\frac{}{\langle ?N : \psi_N(A) \rangle\ \Downarrow (\mathcal{K}\ ?N\ \alpha_1, \ldots \mathcal{K}\ ?N\ \alpha_{d(A)}) : A}\ f^{\mathcal{C}} \in \textit{lindef}(A),\ \alpha_i = \text{rhs}(f^{\mathcal{C}}, i)$$

Here we assume that we know the type of the metavariable $?N$, from the type checker, and that the type $\psi_N(A)$ is mapped to the concrete category A. Once we have the category A, the application of every possible default linearization function in $\textit{lindef}(A)$ is a valid linearization.

2.5.1 Soundness

The linearization is sound if for every transformation $t^{\mathcal{A}} \Downarrow t^{\mathcal{C}} : B$ we can prove that the computed concrete tree is always mapped to the right abstract tree i.e. $\psi_{tr}(t^{\mathcal{C}}) = t^{\mathcal{A}}$ and that $t^{\mathcal{C}} : B$.

In the ABSTRACT TO CONCRETE rule, the computed tree is $f^{\mathcal{C}}\ t_1^{\mathcal{C}} \ldots t_n^{\mathcal{C}}$. By the definition of $P_{f^{\mathcal{A}},\vec{B}}$, we know that $\psi_F(f^{\mathcal{C}}) = f^{\mathcal{A}}$. Also since the rule is recursive we assume by induction that the recursive steps are sound, which give us $\psi_{tr}(t_i^{\mathcal{C}}) = \psi_{tr}(t_i^{\mathcal{A}})$. From the last two we conclude that:

$$\begin{aligned} \psi_{tr}(f^{\mathcal{C}}\ t_1^{\mathcal{C}} \ldots t_n^{\mathcal{C}}) &= \psi_F(f^{\mathcal{C}})\ \psi_{tr}(t_1^{\mathcal{C}}) \ldots \psi_{tr}(t_n^{\mathcal{C}}) \\ &= f^{\mathcal{A}}\ t_1^{\mathcal{A}} \ldots t_n^{\mathcal{A}} \end{aligned}$$

which is a proof of the first part of the soundness condition.

The proof for the second condition is also trivial. The output tree in the rule ABSTRACT TO CONCRETE is always of the right category A because the top-level function $f^{\mathcal{C}}$ is taken from a production which has A as a result category.

2.5.2 Completeness

The linearization rule is complete if it finds all concrete trees $t^{\mathcal{C}}$ for a given abstract tree $t^{\mathcal{A}}$ such that $\psi_{tr}(t^{\mathcal{C}}) = t^{\mathcal{A}}$. The proof is by induction on the tree structure. When the tree is a leaf, i.e. a function without arguments, then by the definition of $P_{f^{\mathcal{A}},\vec{B}}$, we see that we will get all concrete functions $f^{\mathcal{C}}$ such that $\psi_F(f^{\mathcal{C}}) = f^{\mathcal{A}}$. When the top-level function has arguments then we assume that we get all possible concrete trees for the children which in turn give us all possible output trees. This assumption is the induction step, since we get the children from the premisses of the ABSTRACT TO CONCRETE rule.

2.5.3 Complexity

The transformation rule iterates recursively on the structure of the whole tree and at every node there are possibly many choices. The number of choices multiply when you traverse a path from the root of the tree to the leaves. This implies that in the worst case the linearization process is exponential in the depth of the tree. Despite this in practical applications the linearization behaves like a linear process. The explanation for this is that first the depth of the tree is usually small and second the multiple choices manifest only in very special cases.

The only case when we can have nondeterminism is if there are many productions with the same argument categories but a different concrete function or a different result category. The only way to cause this is by using the variants construction in the GF source language. For example we can construct a grammar which for robustness ignores the gender of the nouns. Since it is a common mistake for beginners in a new language to use a wrong gender, it might be a good idea to write a grammar which is more forgiving and accepts both masculine and feminine for some nouns. For instance in Bulgarian, we can define the linearization of the constant apple as:

$$\textbf{lin}\ \mathit{apple} = \{s = \text{"jabălka"};\ g = \textbf{variants}\{\mathit{Fem}; \mathit{Masc}\}\};$$

In this case, the compiler will produce two productions:

$$\mathit{Kind}_{\mathit{Masc}} \rightarrow \mathit{apple}[]$$
$$\mathit{Kind}_{\mathit{Fem}} \rightarrow \mathit{apple}[]$$

At runtime, the linearizer will have to backtrack in order to generate one linearization of the whole sentence – one for feminine and another for masculine.

The common practice is that grammars with a lot of variants are usually not used for linearization but for parsing. In this case, the variants gives us robustness but at the same time cause ambiguities during the linearization. Even when such grammars are actually used for linearization then the default linearization of every variant should be listed first. The interpreter is able to enumerate all possibilities but the user is free to stop the enumeration after the retrieval of the first version which always uses the default linearization[4].

2.6 Literal Categories

GF is not a general purpose language but a language which is specialized for the development of natural language grammars. This means that in some cases we either do not want or we cannot implement all kinds of processing directly in GF. In many cases, we can preprocess the input or postprocess the output from GF to achieve the desired effect. In other cases, however, the additional processing

[4] In lazy languages like Haskell, we actually return the list of all linearizations but the laziness guarantees that they will not be computed until they are needed. Ideally implementations in other languages should use iterator like interface which ensures that the computations are done on demand.

interplays with the grammatical context in a complex way. In such situations, it is helpful if the parser in GF can cooperate with external components written in general purpose programming languages.

A trivial example of such interplay is the way in which the literal categories are supported in GF. So far we only talked about abstract categories whose values are composed algebraically by combining a fixed number of functions. There are three literal categories which are exceptions to the rule:

- String - a category whose values are arbitrary Unicode strings
- Int - a category for integers
- Float - a category for floating point numbers

The common in all cases is that the set of values for the literal categories is not enumerated in the grammar but is hard-wired in the compiler and the interpreter. The linearization rule is also predefined, for example, if we have the constant 3.14 in an abstract syntax tree, then it is automatically linearized as the record $\{s =$ ”3.14”$\}$. Similarly, if we have the string ”John Smith” then its linearization is the wrapping of the string in a record, i.e. $\{s =$ ”John Smith”$\}$.

Now we have a problem because the rules in Section 2.3 are not sufficient to deal with literals. Furthermore, while usually the parser can use the grammar to predict the scopes of the syntactic phrases, this is not possible for the literals since we allow arbitrary unrestricted strings as values of category *String*. Let say, for example, that we have a grammar which represents named entities as literals, then we can represent the sentence:

> John Smith is one of the main characters in Disney’s film Pocahontas.

as an abstract syntax tree of some sort, for instance:

> *MainCharacter* ”John Smith” ”Disney” ”Pocahontas”

This works fine for linearization because we have already isolated the literals as separated values. However, if we want to do parsing, then the parser will have to consider all possible segmentations where three of the substrings in the input string are considered literals. This means that the number of alternatives will grow exponentially with the number of String literals. Such exponential behaviour is better to be avoided, and in most cases, it is not really necessary.

Our solution is that by default every *String* literal is exactly one token, and the *Int* and *Float* literals are recognized according to the syntactic conventions in GF.

If this is not suitable, then the user can provide an application specific recognizer, written in the host language i.e. Haskell, Java, C, etc. Our simplification also implies that some abstract trees can be linearized but after that the output will not be parseable, if there are string literals which are not exactly one token long.

The application specific recognizer has access to the input sentence, and if there is a phrase at the current position which looks like a literal value, then it returns the pair $(\vec{\omega}, t)$ to the parser. Here $\vec{\omega}$ is a tuple of strings with the dimensionality of the literal category and t is the abstract syntax tree that have to be assigned as literal value. The parser handles the feedback from the recognizer by using a new deduction rule:

LITERAL

$$\frac{[^k_j A \to f[\vec{B}]; l : \alpha \bullet \{d; r\}\ \beta]}{N \to g[] \quad [^{k+|\omega_r|}_j A \to f[\vec{B}\{d := N\}]; l : \alpha\ \{d; r\} \bullet \beta]} \qquad \begin{array}{l} g = (\omega_1, \ldots, \omega_{d(\vec{B}_d)}), \\ \psi_F(g) = t \end{array}$$

The new rule is a combination of the existing PREDICT, COMPLETE and COMBINE rules which are now merged because the recognition of the literal category is done outside the parser. The active items in the premises of PREDICT and LITERAL (see Figure 2.2 for reference) are very similar. The only difference is that in LITERAL we have used the new notation $\{d; r\}$ instead of $\langle d; r \rangle$. The compiler generates this new kind of pair when the referenced category is a literal category, and the parser uses this to decide when to call LITERAL and when to call PREDICT.

The difference with PREDICT is that LITERAL does not require the existence of an appropriate production. Since the literal values are not enumerated in the grammar we cannot expect that a production will ever be found. Instead we generate a fresh function symbol g and a fresh category N which together comprise the production $N \to g[]$, which when combined with the item in the premise gives the item in the conclusion.

This is the only rule where we generate new function symbols on the fly. The key for efficient parsing in Section 2.3 was that the parser is able to generate fresh categories which are specializations of existing categories. This time we generate fresh functions, and this lets us handle literals. The intuition for this is that the literal values are not enumerated in the grammar but can be added on demand once they are identified by the external recognizer. The LITERAL rule is fired only the first time when some of the constituents have to be recognized, after that the literal category B_d is replaced with the new category N and the application of the usual PREDICT rule is enough. The rule is simple: if the next symbol after the dot is $\langle d; r \rangle$ then always the PREDICT rule applies. If the next symbol is $\{d; r\}$, then

```
parse :: PGF -> Language -> Type -> Maybe Int -> [Token] -> ParseOutput
parse pgf lang typ dp toks = loop (initState pgf lang typ) 0 toks
  where
    loop ps n []     = fst $ getParseOutput ps typ dp
    loop ps n (t:ts) = case nextState ps (inputWithNames (t:ts)) of
                         Left  es -> ParseFailed n
                         Right ps -> loop ps (n+1) ts

    inputWithNames = mkParseInput pgf lang
                                  tok
                                  [(mkCId "String", name)]
      where
        tok (t:ts) = Map.lookup (map toLower t)
        tok _      = const Nothing

        name ts = let nts = takeWhile isNameTok ts
                  in if null nts
                       then Nothing
                       else Just (mkStr (unwords nts),nts)

        isNameTok (c:cs) | isUpper c = True
        isNameTok _                  = False
```

Figure 2.7: An example for parser with a naïve recognizer for named entities

if the target category is defined in the grammar then the LITERAL rule is applied, and if the category is freshly generated then the PREDICT rule is used.

Figure 2.7 shows an example Haskell code which combines parsing with GF grammar complemented with a simple named entities recognizer. The parser calls `initState` in the beginning, and after that calls `nextState` in a loop for each token. Finally `getParseOutput` is called to get the list of abstract syntax trees. The input to `nextState` is not directly a token but a structure of type *ParseInput* which contains the current recognizer. Whenever a *String* literal has to be recognized, the recognizer checks whether there is a sequence of tokens starting with a capital letter at the current position. If there is, then it is considered as a potential named entity[5]. Independently from the feedback from the recognizer the parser will also consider all alternative analyses based on rules in the grammar. If some phrase is both acceptable as literal and as some other syntactic phrase, then both alternatives are returned as possible. This happens for example if the first word of a sentence can be seen as both name (it starts with capital letter) and a normal token.

Finally we should emphasize that there is nothing specific in the rule LITERAL for the categories *String*, *Int* and *Float*. In principle, the same rule can be fired

[5]any realistic named entities recognizer should use more sophisticated strategy

for any category, and the parser chooses when to fire LITERAL and when PREDICT based on the next symbol after the dot. The compiler will produce either $\{d; r\}$ or $\langle d; r\rangle$ based on command line settings which characterize a given category as a literal or not. Just as well we can setup any category as a literal category.

Again this is best illustrated with an example, for instance, we may want to distinguish between names of people and names of organizations. We can define two categories *Organization* and *Person* and two functions:

$$\begin{aligned} \textbf{fun}\ & \mathit{mkOrganization} : \mathit{String} \rightarrow \mathit{Organization}; \\ & \mathit{mkPerson} \quad\quad : \mathit{String} \rightarrow \mathit{Person}; \end{aligned}$$

which create the corresponding category from a string. This gives us a way to distinguish between the two types of entities but somehow we should communicate to the named entity recognizer what kind of entity is expected. Just attaching a recognizer to the category *String* does not help. Fortunately, by using the option `-literal`, we can tell the GF compiler that we want to have custom literal categories i.e.:

```
> gf -make -literal=Person,Organization MyGrammar.gf
```

After that the parser will treat *Person* and *Organization* as literal categories. The implementation of the named entity recognizer should ensure that in the returned abstract syntax tree the name is wrapped with either function *mkOrganization* or function *mkPerson*.

2.7 Higher-Order Abstract Syntax

In the definition of an abstract syntax (Definition 2, Section 2.1), we defined the allowed types as:

> The type is either a category $C \in N^{\mathcal{A}}$ or a function type $\tau_1 \rightarrow \tau_2$ where τ_1 and τ_2 are also types.

This implies that if τ_1, τ_2 and τ_3 are types, then we also permit $(\tau_1 \rightarrow \tau_2) \rightarrow \tau_3$ as a type. For instance we can define functions which act like logical quantifiers:

$$\textbf{fun}\ \mathit{forall} : (\mathit{Id} \rightarrow \mathit{Prop}) \rightarrow \mathit{Prop}$$

This is a completely acceptable definition in GF but so far we simply ignored that possibility. GF implements the notion of higher-order abstract syntax [Pfenning and Elliot, 1988] which is a handy way to implement bound variables. For

example, we can use the above function in the abstract syntax of sentences which quantify over sets:

For every integer X, it is true that either X is odd or X is even.

the first appearance of X binds the variable to denote an arbitrary integer, and in the second and the third appearance the variable is used to state some property for it[6]. The same sentence can also be modelled by using the literal category *String* but if it is modelled with literals, then the parser will not control whether all variables are bound, i.e. it will also accept:

For every integer Y, it is true that either X is odd or X is even.

where X is not defined anymore. With the high-order function *forall* we can assign the following abstract tree:

$$\mathit{forall}\ (\backslash X \rightarrow \mathit{or}\ (\mathit{odd}\ X)\ (\mathit{even}\ X))$$

and since both parsing and linearization accept only closed lambda terms, the tree will be valid only if all variables are bound.

Before we proceed with the formal rules for parsing and linearization with higher-order abstract syntax, we should clarify the meaning of the higher-order arguments, i.e. arguments of type $A \rightarrow B$, from grammatical point of view. For instance, what does it mean to have such type as a start category (type) in a grammar? In principle it is not very different from having B as a start category. Indeed for every abstract syntax tree t of type B we can build an abstract syntax tree $\backslash x \rightarrow t$ of type $A \rightarrow B$ by just wrapping the original tree t with a lambda abstraction for some fresh variable x. The difference is that when we have function types, then during the parsing of B we should also allow every phrase of category A to be replaced with a variable name. In other words, every time when we have to parse with $A \rightarrow B$ we actually parse with B but we also temporarily add new productions for all concrete categories $A^{\mathcal{C}}$ such that $\psi_N(A^{\mathcal{C}}) = A$:

$$A^{\mathcal{C}} \rightarrow f[\mathit{Var}^{\mathcal{C}}], \quad f \in \mathit{lindef}(A^{\mathcal{C}})$$

Here $\mathit{Var}^{\mathcal{C}}$ is an internal literal category whose values are all tokens conformant with the lexical rules for an identifier in GF. In the previous section, we introduced three built-in literal categories, and now we add another one – $\mathit{Var}^{\mathcal{C}}$. The

[6] Another application of higher-order abstract syntax is to model anaphoric expressions in natural language, see Ranta [2011], Section 6.10, pp. 141-143

difference is that while both *String*, *Int* and *Float* are abstract categories with corresponding concrete categories, $Var^{\mathcal{C}}$ is defined only in the concrete syntax, and it does not have corresponding abstract counterpart. This makes $Var^{\mathcal{C}}$ internal and accessible only for the interpreter because the concrete categories are completely hidden from the users. This also implies that $\psi_N(Var^{\mathcal{C}})$ is undefined and should never be used.

Unfortunately just adding $A^{\mathcal{C}} \rightarrow f[Var^{\mathcal{C}}]$ is not enough to guarantee that all variables are bound since once we have the production, it can be applied indefinitely many times. It turns out that it is far too complicated for the parser to maintain the correct scope in all cases. For this to work, for every item, we have to keep the current scope, and the problem is that it has unlimited depth. For instance when we have functions like *forall*, then we can nest the function and introduce unlimited number of variables, i.e.:

$$forall\ (\backslash X \rightarrow forall\ (\backslash Y \rightarrow forall\ (\backslash Z \rightarrow \ldots)))$$

For precise scope management, we need, for every active item, to keep track of at least the number of variables per category. Since the number of variables is unlimited this could lead to potentially infinite number of items.

A far simpler solution is to say that every time when the parser has to predict new productions, it is also allowed to predict productions which introduce variables, and this is regardless of whether or not the current scope allows it at this point. This means that the final parse chart can contain abstract trees which are not closed, i.e. have unbound variables, but this is not necessarily a problem because when the trees are extracted from the chart, they are typechecked, and the type checker will reject any tree which has unbound variables. After all in the final result only the well-formed trees will be kept.

The problem with this solution is that for every category we must allow one extra production which furthermore is very permissive, i.e. almost every token in the sentence can be interpreted as an instance of almost every category. This immediately affects the parsing performance since at every step the parser has to consider a lot more possibilities even if the grammar does not have any higher-order functions. This is very unfortunate especially given the fact that in typical GF grammars most of the categories are not used as higher-order arguments, so the overhead will be unnecessary.

As a compromise, our solution is to have only partial control over when new variables can be introduced. It can still happen that the parser will generate trees with unbound variables but in a lot fewer cases. In return, the parsing is still

efficient and guaranteed to terminate. The trick is that new productions should be predicted only if they can be potentially used.

The key is a modest extension to the structure of PMCFG productions which explicitly encodes the high-order types. For instance, if we have a function h with a higher-order type:

$$h : (A \to B) \to C$$

then we introduce a new kind of production:

$$C^{\mathcal{C}} \to h^{\mathcal{C}}[A^{\mathcal{C}}/\mathit{Var}^{\mathcal{C}} \to B^{\mathcal{C}}]$$

where $A^{\mathcal{C}}$, $B^{\mathcal{C}}$, $C^{\mathcal{C}}$ and $h^{\mathcal{C}}$ are as usual the corresponding concrete counterparts, but the new thing is the pair $A^{\mathcal{C}}/\mathit{Var}^{\mathcal{C}}$ which encodes the fact that when we predict with category $B^{\mathcal{C}}$, then in addition we have to add the production $A^{\mathcal{C}} \to f[\mathit{Var}^{\mathcal{C}}]$. We do not need f in the production for $h^{\mathcal{C}}$ because we can always look it up from the grammar, e.g. any function in the set $\mathit{lindef}(A^{\mathcal{C}})$ is applied. At the same time, we do need $A^{\mathcal{C}}/\mathit{Var}^{\mathcal{C}}$ instead of just $A^{\mathcal{C}}$ because in the parsing process, once we know the name of the bound variable we can replace $\mathit{Var}^{\mathcal{C}}$ with some fresh category which fixes the name of the variable. More formally, our extension is that the arguments to the productions are not simple categories anymore but rich structures that we call concrete types:

Definition 6 *A concrete type $\tau^{\mathcal{C}}$ is either:*

- *a concrete category C, or*
- *a type $C_1/C_2 \to \tau'^{\mathcal{C}}$ where C_1 and C_2 are some concrete categories and $\tau'^{\mathcal{C}}$ is another concrete type.*

The other extension is that we need to know the variable name in the linearization of functions like *forall*. From user's point of view, the linearization category for arguments of function type, i.e. $\mathit{Id} \to \mathit{Prop}$ for instance, is a record which has the same fields as the linearization of the target category, i.e. *Prop*, but is also extended with the special fields $\$0, \$1, \ldots$ of type *Str* which store the names of the bound variables. For example, the linearization for *forall* is:

lin *forall* $p =$ "for every integer" $++\ p.\$0\ ++$ "it is true that" $++\ p.s$;

where $p.\$0$ is the name of the bound variable and $p.s$ is the linearization of the argument itself, i.e. the proposition $(\mathit{or}\ (\mathit{odd}\ X)\ (\mathit{even}\ X))$ from the example.

LITERAL VAR

$$\frac{[^k_j A \to f[\vec{B}]; l : \alpha \bullet \langle d; \$r \rangle \ \beta]}{N \to g[] \quad [^{k+1}_j A \to f[\vec{B}\{d, r := N\}]; l : \alpha \ \langle d; \$r \rangle \bullet \beta]} \ g = (\nu), \ \psi_F(g) = \nu$$

PREDICT VAR

$$\frac{B_{d,r} \to g[\vec{C}] \qquad [^k_j A \to f[\vec{B}]; l : \alpha \bullet \langle d; \$r \rangle \ \beta]}{[^k_k B_{d,r} \to g[\vec{C}]; 1 : \bullet\gamma]} \ \gamma = \text{rhs}(g, 1)$$

COMBINE VAR

$$\frac{[^u_j A \to f[\vec{B}]; l : \alpha \bullet \langle d; \$r \rangle \ \beta] \qquad [^k_u B_{d,r}; 1; N]}{[^k_j A \to f[\vec{B}\{d, r := N\}]; l : \alpha \ \langle d; \$r \rangle \bullet \beta]}$$

Figure 2.8: Parsing Rules for Variables

In PGF the extra fields are accessed by using a new kind of argument references, i.e. we use $\langle d; \$r \rangle$ as a reference to the r-th bound variable in the argument with index d. The normal constituents are still referenced with pairs like $\langle d; r \rangle$ while the new notation $\langle d; \$r \rangle$ helps us to distinguish the variable references.

With this changes in the PMCFG representation, we are ready to start with the parsing rules for higher-order syntax. The three rules LITERAL VAR, PREDICT VAR and COMBINE VAR on Figure 2.8 implement all cases where the dot is in front of some variable reference. The new notation there is that if $\vec{B}$ is some vector of arguments then $B_{d,r}$ is the literal category for the r-th bound variable in argument with index d. Similarly $\vec{B}\{d, r := N\}$ means that we update the category for the r-th variable in argument with index d.

The rule LITERAL VAR is similar to the rule LITERAL from Section 2.6 except that now we update a category for variable instead of the usual argument categories. Otherwise it functions in exactly the same way, we call the external recognizer for literals and if a variable name is encountered, a new function g and a new production $N \to g[]$ are generated which store the name. At the end the variable category is updated with the new category N.

In the example with the *forall* function the rule LITERAL VAR will be activated after the consumption of the prefix “For every integer”. At this point we have the

item:

$$[^3_0 \mathit{Prop} \rightarrow \mathit{forall}[\mathit{Id}/\mathit{Var}^{\mathcal{C}} \rightarrow \mathit{Prop}]; 1 : \ldots \bullet \langle 1; \$1 \rangle \ldots]$$

Since the next token is "X", and it conforms to the syntax for identifiers in GF, the parser will accept it and will produce two new items:

$$N \rightarrow g[] \qquad [^4_0 \mathit{Prop} \rightarrow \mathit{forall}[\mathit{Id}/N \rightarrow \mathit{Prop}]; 1 : \ldots \langle 1; \$1 \rangle \bullet \ldots]$$

here the second item is derived from the premise by moving the dot to the next position and by replacing the variable category with a new category N. The other generated item is a production for N where the newly generated function g:

$$g := (\text{"X"})$$

stores the name of the variable.

The other two rules PREDICT VAR and COMBINE VAR are similar variations of PREDICT and COMBINE and they are applied when the variable category is not a literal.

In addition to the three new rules, we also need to modify the existing PREDICT rule which will add the new productions. In the new version of PREDICT we now generate more than one item:

$$\text{PREDICT} \quad \frac{D \rightarrow g[\vec{C}] \qquad [^k_j A \rightarrow h[\vec{B}]; l : \alpha \bullet \langle d; r \rangle\ \beta]}{L \rightarrow f[M] \qquad [^k_k D \rightarrow g[\vec{C}]; r : \bullet \gamma]} \ \gamma = \text{rhs}(g, r)$$

here we assume that $B_d = L/M \rightarrow D$ and we predict not only a new active item but also one production for every function $f \in \mathit{lindef}(L)$. If B_d had more high-order arguments, i.e. if it was $L_1/M_1 \rightarrow \ldots L_n/M_n \rightarrow D$, then we must add one production for every pair of categories L_i and M_i.

In our particular example we will derive the item:

$$[^9_0 \mathit{Prop} \rightarrow \mathit{forall}[\mathit{Id}/N \rightarrow \mathit{Prop}]; 1 : \ldots \bullet \langle 1; 1 \rangle \ldots]$$

after we have accepted the prefix "For every integer X, it is true that". At that point, the dot is in front of $\langle 1; 1 \rangle$ and we have to apply the rule PREDICT. Since we have a higher-order argument, in addition to the active item, we will derive the production:

$$\mathit{Id} \rightarrow f[N]$$

where f is the default linearization function for category *Id*.

Note that now we have the category N instead of $Var^{\mathcal{C}}$. This means that during the parsing of the proposition "either X is odd or X is even", the parser will suggest only X and will not accept any other variable name. This ensures that only variable names that are already in the scope will be accepted.

Unfortunately our trick works well only in one direction. If we instead had another linearization for *forall*:

$$\textbf{lin}\ \textit{forall}\ p = \text{"it is true that"} +\!\!+\ p.s\ +\!\!+ \text{"for every integer"} +\!\!+\ p.\$0;$$

then the parser will apply PREDICT before LITERAL VAR since $p.s$ is used before $p.\$0$. In this case the temporary production will be:

$$\textit{Id} \rightarrow f[\textit{Var}^{\mathcal{C}}]$$

and the parser will have to accept as valid any variable name. Furthermore when the parser has to recognize the last appearance of the variable, i.e. the one corresponding to $p.\$0$, it has no clue about what variable names were used in the phrase for $p.s$ so it must again accept any variable name. This still does not mean that unbound variables are permitted because the next step after the pure parsing is the type checking and the type checker will reject any abstract tree which has unbound variables. The only visible effect from the difference is when the parser is used in predictive mode, then in the first case the user will get the correct predictions for variable names while in the second the predictions are more liberal but at the end if the tree is not well formed, the user will get an error message.

From linearization perspective, the linearization of tree of type $A \rightarrow B$ is similar to the linearization of tree of type B, except that whenever we reach a variable, we have to wrap it with some function f in the set of the default linearizations for A.

The precise algorithm for the extraction of trees with higher-order abstract syntax, we will defer until Section 3.5 where we will consider it together with the treatment for dependent types.

2.8 Optimizations

We experiment with grammars which have hundreds of thousands of productions. Although this does not necessary affect the parsing and the linearization efficiency, without proper optimization of the grammar representation, the required memory

will be so high that the usage of the grammar becomes impractical. Indeed, back in year 2008, this was the case with most of the resource grammars. Only after a number of automatic and manual optimization techniques were developed, the direct usage of the grammars was made possible. Before that the resource grammars were usable only as libraries for defining small application specific grammars.

The two main techniques for reducing the grammar size are Common Subexpressions Elimination (Section 2.8.1) and Dead Code Elimination (Section 2.8.2). Both optimizations are implemented as part of the GF compiler but they also affect the grammar representation in PGF, so they are also important for the general understanding of the GF engine.

The optimization in Section 2.8.3 does not affect the representation of the GF grammar but greatly improves the performance of the parser when working with large coverage lexicons (~ 40000 lemmas). Finally in Section 2.8.4 we will describe some techniques for manual optimizations.

2.8.1 Common Subexpressions Elimination

The first important observation for every non-trivial grammar is that there are a lot of repetitions in the definitions of the concrete functions. The concrete functions are compiled from the linearization rules in the original grammar, where every function corresponds to one particular instantiation of the parameters in the original rule. The consequence is that there will be many functions which are almost the same. If we take as an example the linearization:

$$
\begin{array}{l}
\textbf{lin}\ f\ x\ y = \{ \\
\quad s1 = \textbf{case}\ x.p\ \textbf{of}\ \{\mathit{True} \Rightarrow x.s1;\ \mathit{False} \Rightarrow \text{"a"}\}; \\
\quad s2 = x.s2 +\!\!+\ y.s; \\
\quad \}
\end{array}
$$

then when it is compiled to PMCFG, we will get two different concrete functions:

$$
\begin{aligned}
f_1 &:= (\langle 1;1\rangle, \langle 1;2\rangle\langle 2;1\rangle) \\
f_2 &:= (\text{"a"}, \langle 1;2\rangle\langle 2;1\rangle)
\end{aligned}
$$

The functions are different but actually differ only in the first constituent because the value of the parameter $x.p$ does not affect the second constituent. This is a very common situation, and for instance in the resource grammars there are functions with hundreds of constituents where only few differ from function to function.

In the other parts this chapter, we show the constituents as parts of the function definition but in the actual PGF representation we have a separated table which stores all unique constituents. The function definition itself has only pointers to the table. Technically, the right representation for function f in PMCFG would be:

$$\begin{aligned} f_1 &:= (s_1, s_2) \\ f_2 &:= (s_3, s_2) \\ \\ s_1 &:= \langle 1; 1 \rangle \\ s_2 &:= \langle 1; 2 \rangle \langle 2; 1 \rangle \\ s_3 &:= \text{"a"} \end{aligned}$$

Here we have explicit names to the constituents (s_1,s_2 and s_3) which let us to share the sequence $\langle 1; 2 \rangle \langle 2; 1 \rangle$ which would be otherwise duplicated in two places.

Furthermore, there are cases when a naïve compiler would generate multiple copies of exactly the same concrete function. This happens when different instantiations of the parameters in the linearization rules produce the same linearizations.

This can be illustrated with an abstract function with two arguments:

$$\textbf{fun}\ g : C \rightarrow C \rightarrow S$$

which has the linearization:

$$\begin{aligned} &\textbf{lincat}\ S = \{s : \mathit{Str}\}; \\ &\qquad\quad C = \{s : \mathit{Str}; p : \mathit{Bool}\}; \\ &\quad \textbf{lin}\ g\ x\ y = \ \{s = x.s +\!\!+\ y.s \\ &\qquad\qquad\qquad +\!\!+\ \textbf{case}\ \langle x.p, y.p \rangle\ \textbf{of}\ \{ \\ &\qquad\qquad\qquad\qquad \langle \mathit{True}, \mathit{True} \rangle \Rightarrow \text{"eq"}; \\ &\qquad\qquad\qquad\qquad \langle \mathit{False}, \mathit{False} \rangle \Rightarrow \text{"eq"}; \\ &\qquad\qquad\qquad\qquad _ \qquad\qquad\quad\ \Rightarrow \text{"neq"}; \\ &\qquad\qquad\qquad\quad \} \\ &\qquad\qquad\quad \}; \end{aligned}$$

It is obvious that whenever the parameters $x.p$ and $y.p$ are equal, the function will be linearized in one way, i.e. with the word "eq" after the linearizations of $x.s$

and $y.s$, and in another way when the parameters are different. A naïve compiler would generate four concrete functions, i.e. one for every combination of parameter values. However the functions for cases $\langle \mathit{True}, \mathit{True} \rangle$ and $\langle \mathit{False}, \mathit{False} \rangle$ and similarly $\langle \mathit{True}, \mathit{False} \rangle$ and $\langle \mathit{False}, \mathit{True} \rangle$ are exactly the same. Instead of four, the real compiler will produce only two functions:

$$
\begin{aligned}
&S \to g_1[C_{\mathit{True}}, C_{\mathit{True}}] \\
&S \to g_1[C_{\mathit{False}}, C_{\mathit{False}}] \\
&S \to g_2[C_{\mathit{True}}, C_{\mathit{False}}] \\
&S \to g_2[C_{\mathit{False}}, C_{\mathit{True}}] \\
\\
&g_1 := (s_1) \\
&g_2 := (s_2) \\
\\
&s_1 := \langle 1; 1 \rangle \ \langle 2; 1 \rangle \ \text{"eq"} \\
&s_2 := \langle 1; 1 \rangle \ \langle 2; 1 \rangle \ \text{"neq"}
\end{aligned}
$$

Note that now we have two productions that use one and the same function symbol. This requires that every reasonable implementation of the engine should permit the reuse of functions in multiple productions.

The last observation is that it is common to have groups of productions with the same function symbol and result category but different arguments, for example:

$$
\begin{array}{llll}
A \to f[B, C_1, D_1] & A \to f[B, C_2, D_1] & A \to f[B, C_3, D_1] & A \to f[B, C_4, D_1] \\
A \to f[B, C_1, D_2] & A \to f[B, C_2, D_2] & A \to f[B, C_3, D_2] & A \to f[B, C_4, D_2]
\end{array}
$$

In this particular case the only difference between the productions is the choice of categories C_i and D_j i.e. the different productions correspond to different choices of indices i and j. One simple case when this happens is when some parameter is not used at all in the linearization rule. This situation and the solution is already described in Ljunglöf [2004]. The list of productions are compressed by

introducing special coercion productions:

$$\begin{aligned} &A \rightarrow f[B, C, D] \\ &C \rightarrow _[C_1] \\ &C \rightarrow _[C_2] \\ &C \rightarrow _[C_3] \\ &C \rightarrow _[C_4] \\ &D \rightarrow _[D_1] \\ &D \rightarrow _[D_2] \end{aligned}$$

Here the special wildcard function _ marks the coercions.

The optimization described in Ljunglöf [2004] does not capture all cases which can be optimized. In his solution, the fact that some parameter is not used is detected during the generation of the concrete function itself. Unfortunately this is not always so simple. If we take this modified version of the linearization for function g:

$$\begin{aligned} \textbf{lin}\ g\ x\ y = \ &\{s = x.s +\!\!+\ y.s \\ &\quad +\!\!+\ \textbf{case}\ \langle x.p, y.p\rangle\ \textbf{of}\ \{ \\ &\qquad \langle \mathit{True}, \mathit{True}\rangle \Rightarrow \text{"true"}; \\ &\qquad _ \qquad\qquad\quad\ \Rightarrow \text{"false"}; \\ &\quad \} \\ &\}; \end{aligned}$$

then both parameters $x.p$ and $y.p$ are still used but the value of $y.p$ is irrelevant if $x.p = \mathit{True}$ and vice versa. This is not easy to detect during the compilation. We still use Ljunglöf's compilation schema including his optimization but after that, if it is possible, we also merge some productions by using extra coercions.

The introduction of coercions does not change significantly the basic parsing and linearization algorithms. The coercion functions can be seen as identity functions which map one category into another. However, the parsing and the linearization algorithms treat the coercions as special kinds of productions because in this way we do not need extra functions in the concrete syntax of the grammar.

All subexpression elimination optimizations have been implemented and tried with the resource grammar library [Ranta, 2009], which is the largest collection of grammars written in GF. The produced grammar sizes are summarized in Table 2.1. The columns show the grammar size in the number of PMCFG productions,

Language	Productions		Functions		Constituents		File Size (Kb)	
	Count	Ratio	Count	Ratio	Count	Ratio	Size	Ratio
Bulgarian	3521	1.03	1308	2.59	76460	1.84	3119	1.91
Catalan	6047	1.23	1489	4.29	27218	3.82	1375	5.34
Danish	1603	1.05	932	1.51	8746	2.09	360	1.55
English	1131	1.03	811	1.34	8607	5.30	409	2.05
Finnish	135213	-	1812	-	42028	-	3117	103.53
French	11520	1.14	1896	6.05	40790	3.63	3719	4.37
German	8208	1.37	3737	2.81	21337	2.47	1546	2.76
Interlingua	2774	1.00	1896	1.29	3843	2.09	245	1.39
Italian	19770	1.17	2019	9.71	39915	3.86	2193	7.76
Norwegian	1670	1.05	968	1.51	8642	2.19	361	1.57
Romanian	75173	1.50	1853	39.19	24222	5.00	1699	21.09
Russian	6278	1.04	1016	4.10	19035	3.17	985	2.18
Spanish	6006	1.23	1448	4.38	27499	3.75	1369	5.36
Swedish	1492	1.03	907	1.44	8837	2.01	352	1.48

Table 2.1: Grammar sizes in number of productions, functions, constituents and file size, for the GF Resource Grammar Library. The ratio is the coefficient by which the corresponding value grows if the optimizations were not applied.

functions, constituents and in file size. Each column has two subcolumns. The first subcolumn shows the count or the size in the optimized grammar and the second subcolumn shows the ratio by which the grammar grows if the optimizations were not applied. As it shows the optimizations could reduce the file size up to 103 times for Finnish. Without optimizations the compiled Finnish grammar grows to 315 Mb and cannot even be loaded back in the interpreter on a computer with 2Gb of memory.

2.8.2 Dead Code Elimination

Dead code elimination is in general an optimization that removes code that does not affect the execution of the program. In the context of GF, by execution we mean either parsing or linearization with the grammar.

There are well known optimizations for context-free grammars [Hopcroft and Ullman, 1979] which remove useless or unreachable productions in a way preserving the strong generative capacity of the grammar. The same algorithms can be generalized to PMCFG and this is the basis of the dead code elimination opti-

mization in the GF compiler.

It was first observed by Boullier [1998] that every simple Range Concatenation Grammar (sRCG) can be optimized by removing “all its useless clauses by a variant of the classical algorithm for CFGs”. Since sRCG is similar but weaker than PMCFG formalism, it is not surprising that the algorithm for removal of useless productions also generalizes to PMCFG. Unfortunately, Boullier did not describe the algorithm itself, giving the impression that the generalization must be trivial. The first sketch of the actual algorithm was published twelve years later in Kallmeyer [2010][7] who also described the algorithm for the removal of unreachable clauses in sRCG.

Here we describe algorithms for the removal of useless and unreachable productions in PMCFG. The algorithm for removal of useless clauses is basically the same as the one in Kallmeyer [2010] but the algorithm for detection of unreachable code is extended further and can also detect cases when only some of the constituents of a category are unreachable. If the same algorithm is applied to sRCG this will correspond to the removal of arguments for some predicates.

The optimizations were evaluated as part of the project described in Enache and Détrez [2010] where a variant of the GF engine is used for running a touristic phrasebook for fifteen languages on Android phones. Since in this case the applications have to run on a platform with limited resources it is important to make the grammars as small as possible. Thanks to the optimization the whole grammar was reduced from 15Mb to just 0.5Mb. The main source of redundancy was that while the resource library provides full inflection tables and all grammatical tenses, in the phrasebook grammar many morphological forms and most of the tenses are not used. Again the optimizations are the most effective for Finnish where they eliminated many word forms from the otherwise rich Finnish morphology.

Useless Productions

A production is useless if it cannot be used for the derivation of any string in the language. The simplest example is when we have a production whose argument categories do not have productions, i.e. we have:

$$A \to f[B]$$

but we do not have any productions for B and we do not have higher-order productions where B is a higher-order argument. In this case, the production can be

[7] About three months before this section was completed.

safely discarded. Another example is a cyclic chain of productions where there is no production that can be used to break the chain. For instance if we have:

$$A \to f[A]$$

and if this is the only production for A, then we cannot break the loop and the derivation cannot yield a finite string.

The elimination algorithm iteratively computes sets of useful (alive) productions until a fixed point is reached. The initial set is empty:

$$L_0 = \emptyset$$

We compute on each iteration an extension L_{i+1} of the set from the previous iteration L_i which includes the new productions whose arguments already have productions in L_i, i.e.:

$$L_{i+1} = L_i \cup \{A \to f[\vec{B}] \mid A \to f[\vec{B}] \in P;\ B_1, \ldots, B_{a(f)} \in cat(L_i)\}$$

Here by $cat(L_i)$ we mean the set of categories which appear in L_i either as the result category for some production or if $C_1/C_2 \to C_3$ is a higher-order argument, then in the place of C_1. We also add to the set all literal categories, i.e. $\{\mathit{String}, \mathit{Int}, \mathit{Float}, \mathit{Var}\}$, since they are useful despite that we do not have productions for them. The algorithm terminates when $L_{i+1} = L_i$. Since the sequence of sets $L_0, L_1, \ldots$ has an upper limit P and the sequence is monotonous, it is always guaranteed to reach a fix point.

Unreachable Constituents and Productions

When we do parsing, we start from the productions for the start category, and we recursively predict items for other categories. If in this way, there is no way to predict an item for some category or constituent, then they are unreachable.

The reachability of a category is the criterion for discarding productions in the algorithm in Kallmeyer [2010]. Our algorithm is more refined and relies on a more precise criterion that takes into account the reachability of the constituents since when the parser predicts new items, it does it on the basis of a pair of a category and a constituent index.

If there is no way to produce an item for a given pair of a category and a constituent index, then the pair is unreachable. Unfortunately, we cannot simply discard all unreachable constituents of a given category because, according to the

definition of PGF (Section 2.1), all concrete categories for the same abstract category must have the same dimensions i.e. $d(A) = d(\psi_N(A))$. We can maintain this invariant, if instead we compute the reachability for pairs of an abstract category and a constituent index. This means that if we have two categories A_1 and A_2 such that $\psi_N(A_1) = \psi_N(A_2) = A$, then the reachable constituents for A will be those who are reachable for at least A_1 or A_2.

Before computing the reachable pairs, we first compute the set of pairs that are direct neighbours, i.e. there is some production which relates the categories and the constituents:

$$\mathcal{R}el = \{(\psi_N(A), l, \psi_N(B_d), r) \mid A \to f[\vec{B}] \in P,\ rhs(f, l) = \alpha\langle d; r\rangle\beta\}$$

Note that to compute $\mathcal{R}el$ we iterate over the productions in the concrete syntax but in the output we map every concrete category to its abstract category. In this way, we eliminate the difference between concrete categories related to the same abstract category.

The reachability analysis is again an iterative process. We start with the initial set which consists of only one pair with the start category S and the constituent 1:

$$R_0 = \{(S, 1)\}$$

After that on every iteration we add those pairs that are direct neighbours of the categories that are already in the reachability set.

$$R_{i+1} = R_i \cup \{(B, r) \mid (A, l) \in R_i,\ (A, l, B, r) \in \mathcal{R}el\}$$

The iterations stop when we reach a fixed point i.e. $R_{i+1} = R_i = R$. Since the set of categories and constituents is finite and the progression is monotonous, it is guaranteed that a fixed point will be reached.

Once we have the reachability set R we can eliminate several things from the grammar. First, we need to keep only those productions whose result categories map to abstract categories in the reachability set, and we can discard the rest. Second, we process the productions which are not discarded, and we replace in $A \to f[\vec{B}]$, the function f with a new function which contains only the constituents that are reachable for A. Finally since we replaced all functions with new ones, we have also discarded some constituents. As we mentioned in Section 2.8.1, the constituents itself are kept in a separated table which lets us implement common subexpression elimination. After the reachability analysis, we can reduce the table by discarding the constituents that are not referenced anymore

from any function. In the elimination process, we also update the constituents itself because whenever we have $\langle d; r \rangle$ in some constituent then we have to replace r with r' which is the new index of the same constituent in the optimized version of the grammar.

Soundness

The soundness of the removal of useless productions depends on the interpretation of metavariables in PGF. Let's say that we have some *useless* function:

$$\textbf{fun}\ \textit{useless} : A \rightarrow B;$$
$$\textbf{lin}\ \textit{useless}\ x = \text{"useless"};$$

where we either do not have any productions for category A or every prediction with A will always go back to prediction with B. If we use an unoptimized grammar, then the parsing with category B will successfully recognize the string "useless" and the parser will return the abstract tree:

$$\textit{useless}\ ?0$$

Here we get the metavariable $?0$ because the parser detects that the argument of *useless* is never used and it does not try to substitute it with some value. In this case, we get an abstract tree but it has a metavariable which cannot be refined in any way. After the elimination of useless productions, the situation is different. The function will be detected as *useless* and discarded[8] which means that the parser will fail.

Another situation where the optimized and the unoptimized grammars will have different behaviours is when we use the parser in predictive mode, i.e. the user writes some sentence and at each step the parser guides the user by predicting the set of all possible words at the current position.

In some languages, the subject of a sentence can be in different grammatical cases depending on the type of the verb in the sentence. This can be modelled with a function like:

$$\textbf{fun}\ \textit{Pred} : N \rightarrow V \rightarrow S;$$
$$\textbf{lin}\ \textit{Pred}\ \textit{subj}\ \textit{verb} = \{s = \textit{subj}.s\ !\ \textit{verb}.c \mathbin{+\!+} \textit{verb}.s\};$$

[8] the optimizer looks only at the categories and not in the function definitions

Here the parameter c of the verb selects the right case for the subject. Now, if we assume that the language has only nominative and accusative cases, then the compiler will generate two productions:

$$S \rightarrow Pred_{nom}[N, V_{nom}]$$
$$S \rightarrow Pred_{acc}[N, V_{acc}]$$

where the concrete function $Pred_{nom}$ will attach a subject in nominative case to a verb expecting nominative, and similarly $Pred_{acc}$ will attach a subject in accusative. With these two productions, the parser will predict that every sentence starts with a noun in either nominative or accusative case. However, if it happens that the lexicon does not include any verb expecting accusative, then when the user selects a noun in accusative, on the next step the user will get an empty set of predictions.

This is confusing because the expectation is that the predictions always lead to a complete sentence. The situation is also rather common because the different applications usually reuse syntactic rules from the GF resource library but supply their own lexicon. Since the resource library has to be reusable, it implements all grammatical cases even if some of them are not used in most applications. Fortunately, the elimination of useless productions solves the problem since it detects that the second production for *Pred* is useless if none of the verbs expects an accusative subject. After the removal of the second production the parser will correctly predict that a sentence can only start with a noun in nominative.

The final conclusion is that the optimized and the unoptimized grammars are semantically different. Mostly because of the nicer behaviour of the parser for the optimized grammars, we choose that this will be the default semantics of PGF. A given sentence is considered as described by the grammar only if there is a complete finite abstract syntax tree that can be linearized to the same string. The correct behaviour is ensured by always removing the useless productions at the end of the compilation.

The effect from the removal of unreachable productions and constituents is that with the optimized grammar it is safe to linearize only abstract syntax trees whose type is the same as the start category. For comparison, the GF shell lets the user to linearize trees of any type. Still in many applications the restriction is acceptable and the advantage of having smaller grammars is beneficial. In order to satisfy the requirements of the different applications this optimization is performed only when the grammar is compiled with the flag `-optimize-pgf` i.e.:

```
> gf -make -optimize-pgf MyGrammar.gf
```

2.8.3 Large Lexicons

The improvements in the parsing performance in GF opened the way for experiments in open-domain parsing. While this is still an open problem (Section 4.1) and an active research area, some optimizations were necessary to get started.

The resource grammars in GF are traditionally used as libraries for building smaller application specific libraries, but just as well they can be used directly for parsing and linearization. The problem is that for this to be interesting for open-domain parsing, the grammar has to be complemented with a large enough lexicon. The GF 3.2 distribution comes with large coverage lexicons (more than 40000 lemmas) for Bulgarian, English, Finnish, Swedish and Turkish. Combined with the corresponding resource grammars these lexicons make it possible to analyse almost free text.

The new challenge here is the gigantic gap between the usual sizes of the application specific lexicons, i.e. few hundred lemmas, and the sizes of the open-domain lexicons which have several thousand lemmas. The parser as it was described in the previous sections would be very inefficient in this case because of its top-down design. Starting from the start category, after few prediction steps, the parser will reach to a point where it have to predict with some of the lexical categories, i.e. noun, verb or adjective. Unfortunately, the prediction step is linear in the number of productions per category which in this case means that the parser will have to sequentially traverse 40000 productions for almost every word in the sentence.

Ideally we should apply some bottom-up filtering in the top-down parser and this would eliminate most of the redundant traversals. One complication to this is that our grammar is dynamic, i.e. it is extended in the parsing process, and this leads to the question of how to implement the filtering for the newly generated categories. This is solvable in principle if we retain the relation between the fresh categories and the original categories in the grammar. The filter for the original category can also be used as a filter for its specialization.

Unfortunately in our experiments for bottom-up filtering we observed that this requires building large tables and at the end is not very practical idea. Instead, our lightweight solution is to have a mixed parsing strategy where the syntactic rules are processed in top-down fashion and the lexical rules are processed bottom-up. The strategy is similar to the approach in most state of the art statistical parsers where the parser builds only in the syntactic phrases, while the

terminals are tagged in advanced with preterminals, i.e. part of speech tags.

The GF formalism does not have any intrinsic distinction between lexical and syntactic rules. However, by convention, abstract functions without arguments are seen as abstract words. They are special because the definitions of the corresponding concrete functions cannot have any variables, i.e. they are tuples of plain strings. During the grammar loading, all such concrete functions are split out of the main grammar and are reorganized into a nested map:

$$<N^{\mathcal{C}}, \mathbb{N}> \mapsto (T^* \mapsto F^{\mathcal{C}})$$

which represents the lexicon. Here every pair of a concrete category $B \in N^{\mathcal{C}}$ and a constituent index r $\in \mathbb{N}$ such that $0 < r \leq d(B)$ is mapped into a trie from a sequence of tokens (an element of T^*) to a set of functions which have that sequence in its r-th constituent and are also used in at least one production where B is the result category.

Now when we have the active item:

$$[^k_j A \to f[\vec{B}]; l : \alpha \bullet \langle d; r \rangle \; \beta]$$

then we first apply the usual PREDICT rule which traverses only the rules that are not split out, i.e. only the syntactic rules. In addition, we also look up in the lexicon the pair $<B_d, r>$, and we retrieve the trie. After that we match the current input with the prefixes in the trie, and if there are any matches, the corresponding items are predicted and added in the chart. This simple reorganization of the grammar ensures that the parsing performance will degrade only logarithmically instead of linearly with the size of the lexicon.

2.8.4 Hints for Efficient Grammars

Just like with any other programming language, the automatic optimizations in the GF compiler are not always enough to achieve the best performance. They can only do limited transformations which reduce the size of an already existing grammar. The grammarian, on the other hand, can reorganize the grammar in a much more creative way, and this can lead to improvements that go far beyond the capabilities of the automatic optimizer.

Of course, for this to be fruitful, the grammarian must have some understanding of which constructions are the bottleneck of the grammar. It is not possible to give comprehensive guidelines for all situations, but still we can give some hints for manual grammar optimization which are based on our experience and

understanding of the internals of the compiler. The hints are mostly important for the developers of the resource grammars library because when an application grammar is build using the library, it very much benefits from the efficiency of the underlying resource grammar.

The most performance critical operation in the GF interpreter is the parsing, and although its theoretical complexity is always polynomial to the length of the sentence, the exponent and the constant factor are function of the grammar, so a careful optimization can do a lot. In other cases, the high exponential factor is unavoidable and is caused by the nature of the grammar. Here we concentrate on optimizations in the concrete syntax because the abstract syntax implements some theory of the application domain and is therefore fixed. In most cases, a smaller set of productions leads to better performance, and we can start with some hints for reducing the number of productions.

Remember that the compiler generates one concrete category for every combination of values for the inherent parameters in the linearization type for the abstract category. This means that there is an easy way to predict the maximal number of productions by just looking at the type signatures for the abstract functions and at the linearization types for the abstract categories.

We start with an equation for computing the number of values for a parameter type. If we have the definition of the parameter P:

$$\begin{aligned} \textbf{param}\ P\ &= P_1\ Q_{11}\ Q_{12} \ldots Q_{1m_1} \\ &\mid P_2\ Q_{21}\ Q_{22} \ldots Q_{2m_2} \\ &\quad \ldots \\ &\mid P_n\ Q_{n1}\ Q_{n2} \ldots Q_{nm_n} \end{aligned}$$

where $P_1 \ldots P_n$ are the parameter constructors and Q_{ij} are other parameter types, then we can count the number of values $\mathbb{C}(P)$ for P by computing the equation:

$$\begin{aligned} \mathbb{C}(P)\ &= \mathbb{C}(Q_{11}) * \mathbb{C}(Q_{12}) \ldots \mathbb{C}(Q_{1m_1}) \\ &+ \mathbb{C}(Q_{21}) * \mathbb{C}(Q_{22}) \ldots \mathbb{C}(Q_{2m_2}) \\ &\quad \ldots \\ &+ \mathbb{C}(Q_{n1}) * \mathbb{C}(Q_{n2}) \ldots \mathbb{C}(Q_{nm_n}) \end{aligned}$$

Since the parameters must not be recursive, i.e. P itself must not occur as a type for the arguments of its constructors nor in the definitions of Q_{ij}, the equation is also non-recursive and therefore well-founded. A record type consisting of only

parameter types is another parameter type. The number of values for the record is computed by multiplying the number of values for the fields, i.e.:

$$\mathbb{C}(\{q_1 : Q_1; q_2 : Q_2 \ldots q_n : Q_n\}) = \mathbb{C}(Q_1) * \mathbb{C}(Q_2) \ldots \mathbb{C}(Q_n)$$

Similarly, a table, whose values are parameters, can also be used as a parameter. However, the number of values for the table type is exponential:

$$\mathbb{C}(P \Rightarrow Q) = \mathbb{C}(Q)^{\mathbb{C}(P)}$$

and it is better to avoid the usage of table types as parameters.

The number of concrete categories for a single abstract category is computed by multiplying the number of values for all inherent parameters in its linearization type. For instance if we have the parameter types $Q_1, Q_2 \ldots Q_n$ in the linearization type for abstract category A:

$$\textbf{lincat } A = \{\ldots\ ; q_1 : Q_1;\ q_2 : Q_2; \ldots\ ; q_n : Q_n\};$$

then the number of corresponding concrete categories for A will be:

$$\mathbb{C}(A) = \mathbb{C}(Q_1) * \mathbb{C}(Q_2) * \ldots * \mathbb{C}(Q_n);$$

Note that in all equations we multiply more and more numbers which means that the total counts grow exponentially. Fortunately neither the number of parameters nor the number of concrete categories directly affects the size or the efficiency of the grammar. The set of parameter values, for instance, is not stored anywhere in the compiled grammar and therefore this does not affect the size. The set of concrete categories is stored but since all concrete categories, derived from the same abstract category, are encoded as a consecutive integers, it is enough to store the index of the first and the last category. In this way, the number of concrete categories can grow quickly but the space allocated for their descriptions (See object CncCat in Appendix A) is only proportional to the number of distinct abstract categories which is a constant.

So far, all the computations were exact, i.e. if we have computed that there will be a number $\mathbb{C}(A)$ of concrete categories derived from A, then this is the exact number. In contrast, when we compute the number of productions, without doing the actual compilation, then in the best case we can only hope to do pessimistic estimation, i.e. we compute the maximal number which we get if the compiler is not able to do any optimizations. This is still a useful hint because in some corner cases the full compilation may become very slow or may even be impossible due

to memory limitations. But the pessimistic estimation is very cheap and can be done in advance. The estimation can be a hint for which linearization rules have to be optimized first.

The way to estimate the number of productions generated for the abstract function f with signature:

$$\textbf{fun}\ f : A_1 \rightarrow A_2 \rightarrow \ldots \rightarrow A_n \rightarrow A;$$

is by multiplying the number of concrete categories for every argument i.e.:

$$\mathbb{C}(f) = \mathbb{C}(A_1) * \mathbb{C}(A_2) * \ldots * \mathbb{C}(A_n)$$

The number of parameters in the target category A does not affect the number of productions because the corresponding values are either fixed or they are computed on the basis on the parameters in the arguments.

GF includes a profiler which automatically computes the estimations for all categories and functions. If the grammar compilation is triggered with the command line option `-prof`:

```
> gf -make -prof MyGrammar.gf
```

then the compilation proceeds as usual but in addition the profiler will dump the estimations to the standard output. Although these are only estimations and in many cases the optimizer will be able to significantly reduce the number of productions, looking at the estimations is useful in two ways – first reducing the number of possibilities might help the compiler to find better optimization and second this can improve the compilation time.

The general principle for manual optimization is to minimize the number of parameter values. Let's take for example, the case where the noun and the adjective in a noun phrase have to agree in number and gender. The most straightforward implementation is to introduce two new parameter types - one for number and one for gender:

$$\begin{aligned} \textbf{param}\ & \textit{Number} = \textit{Sg} \mid \textit{Pl}; \\ & \textit{Gender} = \textit{Masc} \mid \textit{Fem} \mid \textit{Neutr}; \end{aligned}$$

Although this is principally correct, it does not reflect the fact that in many languages the adjective has only one form for plural which is not dependent on the gender. In such cases, more appropriate parameter type will be:

$$\textbf{param}\ \textit{NumGen} = \textit{GSg}\ \textit{Gender} \mid \textit{GPl};$$

where the gender is available only for singular. If we count the number of parameter values, then we get six possible values for a pair of *Number* and *Gender* and only four values for *NumGen* which is a good improvement.

Another less common feature that also affects the number of productions in the grammar is the free variation. It is often the case that within a certain domain there are several ways to express one and the same thing. In GF, this is modelled by allowing free variations in the linearization rules. For example, if we have an office assistant where the user can issue speech commands, then we might want to have the abstract command *start_word*:

$$\begin{aligned}\textbf{lin}\ \mathit{start_word}\ = & \textbf{variants}\{\text{"open"};\text{"start"}\} +\!\!+ \\ & \textbf{variants}\{\text{"Word"};\text{"Microsoft Word"}\};\end{aligned}$$

which will correspond to four different concrete phrases. Despite the differences, in the given doman, we may not want to differentiate so we just define them as variants. Now in the concrete syntax there will be four different functions which are mapped to the same abstract function:

$$\begin{aligned}\mathit{start_word}_1 &:= (\text{"open Word"});\\ \mathit{start_word}_2 &:= (\text{"open Microsoft Word"});\\ \mathit{start_word}_3 &:= (\text{"start Word"});\\ \mathit{start_word}_4 &:= (\text{"start Microsoft Word"});\end{aligned}$$

The free variation is a useful tool but it have to be used carefully. Note that when the variants are concatenated, the number of possibilities multiply, so having too many variations could lead to combinatorial explosion. It is even worse when the explosion is not easy to find, for example the variants can be in operations which are only later expanded in the linearization rules. At the same time, if we refactor the abstract syntax and add explicit categories for operation, i.e. start/open, and for product, i.e. Word/Microsoft Word, then we can eliminate the variations all together which usually improves the parsing efficiency. This solution also scales better if we want to add more products, i.e. Open Office. In this case Word is no longer shorthand for Microsoft Word because it can also refer to the Word processor in Open Office.

The variants also interact in an unexpected way with table abbreviations. Let's say that we want to add tenses to the office assistant grammar. The most straightforward solution is to define the linearization of an action like *close_word* as a

table indexed by the grammatical tense[9]:

lin *close_word* $= \backslash\backslash t \Rightarrow$ *close_V* ! t ++ **variants** {"Word"; "Microsoft Word"};
oper *close_V* = **table** *Tense* ["close"; "closed"; "have closed"; ...];

Unfortunately this is not what we want because the construction $\backslash\backslash t \Rightarrow \ldots$ is just an abreaviation for a table where the values in the table are a function of the parameter. This means that our definition will be expanded to:

lin *close_word* = **table** *Tense* [
"close" ++ **variants** {"Word"; "Microsoft Word"};
"closed" ++ **variants** {"Word"; "Microsoft Word"};
"have closed" ++ **variants** {"Word"; "Microsoft Word"}
...
];

Now it is obvious that actually we have three or more variants instead of only one. Every variant has two possible values, which means that in total we will have 2^3 possible concrete functions for *close_word*. This is not what we want because in every sentence where *close_word* appears, it will be used only once with one particular tense, so it would be enough if we had only two versions of *close_word* where one and the same variant is used for all tenses. This can be achieved if we restructure the definition of *close_word* by moving the free variation out of the table abbreviation, for instance:

lin *close_word* = **variants** {*closeIt* "Word"; *closeIt* "Microsoft Word"};
oper *closeIt obj* $= \backslash\backslash tense \Rightarrow$ *close_V*!$tense$ ++ obj;

Now the variants construction has wider scope and this will lead to the generation of only two concrete functions.

Finally, as a final resort for tracking performance issues, it is always possible to examine the productions in the compiled PGF. The compiler will dump the portable grammar format in textual representation if it is asked to produce the output in `pgf_pretty` format:

```
> gf -make -output-format=pgf_pretty MyGrammar.gf
```

Experimenting and looking at the output from the compiler is sometimes the best way to fine tune the grammars.

[9]Here the notation **table** *P* [...] introduce a table indexed by the parameter type *P*. The values in the table are listed in the order in which the corresponding parameter values are defined.

Chapter 3

Reasoning

Grammatical Framework is all about grammars and language but the essence of most formal and natural languages is not only in their syntax but also in the semantics. While in the previous chapter we were mostly concerned with the syntax, here we will concentrate on the semantics.

The common methodology for writing application grammars is to start by designing an abstract syntax which reflects the semantics of the domain and after that to render the abstract structures into natural language by designing a concrete syntax. For most applications, the abstract syntax has only simple categories and functions, and then we do not need anything more than what was already discussed in the previous chapter. In other cases, however, we want to express complex semantic restrictions, and then we have to resort to the full power of the logical framework which is the foundation of the abstract syntax.

Note that even if we keep the abstract syntax simple, this does not mean that it is less semantic. When we have categories like *Kind* and *Quality*, as in the Foods grammar, then we have more precise categorization of the words in the language than the merely syntactic categorization of nouns or adjectives. It is certainly possible to go a long way by just making the categories more and more specific.

When this is not enough or if this would make the grammar too complicated, then it is the right time to start using **dependent types**, i.e. context-dependent categories where the context is encoded with some abstract syntax expressions. The last sentence is a precaution because once the dependent types are introduced, then it is no longer guaranteed that all operations with the grammar will terminate. For instance, while the core parsing algorithm has polynomial complexity, after the parsing, the abstract syntax trees have to be type checked which is undecidable in the presence of dependent types and computable functions. Of course, this is

not a reason not to use dependent types. After all the same applies to all general purpose programming languages and they are still very useful. It only means that the programmer should be careful to write programs that terminate.

The abstract syntax in GF is an implementation of Martin Löf's type theory where for type checking we use the same algorithm as the one applied in Agda [Norell, 2007]. This by itself does not make GF an alternative to Agda since the abstract syntax in GF has restrictions which makes it suitable as a grammar description language but at the same time limits its application as a general purpose language. Before going into details, lets generalize our definition of types in the abstract syntax in a way that permits dependent types:

Definition 7 *A type is one of:*

- $A \rightarrow B$ *is a nondependent function type where* A *and* B *are also types;*
- $(x : A) \rightarrow B$ *is a dependent function type where* x *is a variable of type* A *which is free in the type* B*;*
- $(\{x\} : A) \rightarrow B$ *is a function type with implicit argument where again* x *is a variable and* A *and* B *are types;*
- $C\ e_1\ e_2\ \ldots\ e_n$ *is an atomic type where* C *is a category and* $e_1, e_2 \ldots e_n$ *are expressions.*

Here we still have the construction $A \rightarrow B$ which corresponds to simple function types but we also added the construction $(x : A) \rightarrow B$ which is a function where the type of the result depends on the actual value of the argument[1]. The variable x is bound to the value of the argument and is used in the type B. The other generalization is that now the categories can be indexed with values of some type, i.e. instead of having just the category C as an atomic type, we have $C\ e_1\ e_2\ \ldots\ e_n$ where $e_1, e_2 \ldots e_n$ is a possibly empty sequence of expressions which represents the context. Variables like x, bound by the dependent function type, are used exactly in these expressions and make the atomic types dependent on the values of the arguments.

[1]In the source language of GF, $A \rightarrow B$ is a syntactic sugar for $(x : A) \rightarrow B$ when x does not appear in B. This is a more traditional point of view but internally we chose to treat the two notations as different because this will let us to express two optimizations. First if we know that B is not dependent on the value of A, then we do not need to update the environment for B when we compute types. Second the scheduling of the proof search is different for the different types.

Finally, an orthogonal extension is that, like in Agda, we allow implicit arguments, i.e. function arguments whose values can be inferred from the context, so the user does not have to specify them explicitly. This is an important feature and is supported in most dependently typed languages, since without it even programs of small size become too verbose. If we take as an illustration for dependently typed program the usual example of a vector with fixed length, then in GF it is:

$$
\begin{aligned}
\textbf{cat}\ & \mathit{Nat} \\
& \mathit{Bool} \\
& \mathit{Vec}\ \mathit{Nat} \\
\textbf{fun}\ & \mathit{zero} : \mathit{Nat} \\
& \mathit{succ} : \mathit{Nat} \rightarrow \mathit{Nat} \\
& t, f : \mathit{Bool} \\
& \mathit{nil} : \mathit{Vec}\ \mathit{zero} \\
& \mathit{cons} : (s : \mathit{Nat}) \rightarrow \mathit{Bool} \rightarrow \mathit{Vec}\ s \rightarrow \mathit{Vec}\ (\mathit{succ}\ s)
\end{aligned}
$$

Here *Nat* is a category with two constructors *zero* and *succ* which encode natural numbers in Peano representation. *Vec* is the category for a vector with fixed size containing boolean values (the *Bool* category). The index of *Vec* is the length of the vector represented as a natural number. The empty vector has length *zero* but every time when the constructor *cons* is applied, the length is increased with one. Now if we want to construct the vector $\langle t, f, t \rangle$ of length 3, then the abstract syntax for it will be:

$$\mathit{cons}\ 2\ t\ (\mathit{cons}\ 1\ f\ (\mathit{cons}\ 0\ t\ \mathit{nil}))$$

It is obvious that specifying the length for every application of *cons* is first tedious and second unnecessary since the compiler can infer it. The way to avoid it is to use implicit arguments, i.e. we use the type $(\{x\} : A) \rightarrow B$ instead of $(x : A) \rightarrow B$, and in this way we state that there is an argument of type A but its value is omitted and should be automatically inferred[2]. If we change the signature of *cons* to:

$$\textbf{fun}\ \mathit{cons} : (\{s\} : \mathit{Nat}) \rightarrow \mathit{Bool} \rightarrow \mathit{Vec}\ s \rightarrow \mathit{Vec}\ (\mathit{succ}\ s)$$

[2]The equivalent syntax in Agda is $\{x : A\} \rightarrow B$ but we chose to use different syntax because otherwise there will be a conflict with the syntax for records in GF. Other languages like LEGO, Epigram, Twelf, Coq, etc. have similar notations for implicit arguments.

then the abstract syntax expression for the vector is reduced to:

$$\mathit{cons}\ t\ (\mathit{cons}\ f\ (\mathit{cons}\ t\ \mathit{nil}))$$

Still if we want to explicitly give a value to an implicit argument then we can do it by wrapping the argument in curly braces, i.e. $\mathit{cons}\ \{0\}\ t\ \mathit{nil}$ is exactly the same as $\mathit{cons}\ t\ \mathit{nil}$. The later also brings the issue that we must define an extended notion of abstract syntax expressions:

Definition 8 *An abstract syntax expression is one of:*

- $e_1\ e_2$ *is a normal function application;*
- $e_1\ \{e_2\}$ *is an application where* e_1 *is a function with an implicit argument, and* e_2 *is the explicit value of the argument;*
- $\backslash x \rightarrow e$ *is a lambda abstraction;*
- $\backslash\{x\} \rightarrow e$ *is a lambda abstraction which introduces function with an implicit argument;*
- c*, is a function call where* c *is the function name;*
- $\#i$*, is a local variable where* i *is a natural number representing its de Bruijn index. The local variables are introduced either in a lambda abstraction or in a dependent function type.*
- $?i$ *is a metavariable with index* i

Here we explicitly introduce de Bruijn indices [Bruijn, 1972] because this is what we will use in the computation rules. In the GF language, however, the variables are referenced by name, and this is what we will continue to use in the examples, unless if there is a reason to emphasize on the indices. Note that the variable names are still preserved in the binding sites, i.e. in the function types and the lambda abstractions. This makes it easier to print the abstract syntax expressions in a user friendly style and is also the only way to get the variable names when we linearize higher-order abstract syntax trees to natural language.

With these definitions at hand, we are ready to start the discussion for the logical aspects of the framework. Perhaps the most severe restriction of the abstract

syntax is that all types must be monomorphic. For instance, it is not possible to define the signature of the identity function[3]:

$$\textbf{fun}\ \mathit{id} : (A : \mathit{Type}) \rightarrow A \rightarrow A$$

because this would require parametrization over the type of the domain and the range of the function. This restriction is necessary because otherwise it would not be possible to determine the linearization categories for the polymorphic arguments. In other words, if we wish to give linearization for function *id*, then we need to know the abstract category of every argument in order to find the appropriate linearization category. In the polymorphic case, we do not know it because we have type variables. The restriction is reflected in our definitions where we did not include the possibility that the type can be a variable but we let the variables in the expressions to be bound by either a lambda abstraction or a dependent function type.

In principle, we can avoid the restriction by saying that the linearization category for arguments of variable type is some opaque type and the only thing that we can do is to pass around values of this type. This would let us to write a valid linearization rule for *id*:

$$\textbf{lin}\ \mathit{id}\ _\ x = x$$

but this is the only possible rule since we cannot do anything interesting with x.

The other reason not to have polymorphic types is that the categories guide the parser when and whether a certain rule is applicable. When we have polymorphic types, then functions like *id* can be applied in absolutely every context without restriction. This would significantly increase the parsing complexity.

The monomorphism is the most permissive restriction which ensures that the category of every argument is statically known. Unfortunately, this is not the restriction that is currently applied in GF, instead it is forbidden at all to use the type *Type* in the abstract signatures. This means, for instance, that the definition:

$$\textbf{fun}\ \mathit{showType} : \mathit{Type} \rightarrow S$$

will be rejected despite that it is monomorphic. Technically, this kind of functions should not be rejected, if it was allowed to define a linearization category for *Type*

[3]Here *Type* is the category (the type) of all types except *Type* itself. In other frameworks, and in Agda in particular, the same type is called *Set*.

and linearization rules for the categories. In this way we could have for example:

lincat *Type* = *Str*
lin *S* = "S"

and it would not be a problem to have a linearization rule for *showType*. This is possible in principle but is not supported so we leave it as a potential extension of the language in the future.

When the type signatures are seen as logical formulas, then following the Curry-Howard correspondence, the monomorphism restriction corresponds to the restriction that only first-order formulas are allowed in our logic. Since the categories correspond to logical predicates, the restriction that we cannot have type variables is equivalent to saying that we cannot have variables which range over the set of predicates. The impact of this observation is that the automatic reasoning in GF is a lot easier than in a language like Agda which is based on a higher-order logic. In fact the reasoning in GF is reduced to running a Prolog program.

Furthermore, since type theory is an intuitionistic logic, we can classify the logical fragment of GF as a first-order intuitionistic logic. Generally speaking the intuitionistic logic constrains the choice of proof search procedures because there is no intuitionistically equivalent conjunctive normal form for an arbitrary logical formula. Fortunately, in our case, the only logical connective that we allow is implication and in the common case the higher-order syntax is rarely used, so we have mostly simple types like:

$$A_1 \rightarrow A_2 \rightarrow \ldots A_n \rightarrow A$$

which logically correspond to Horn clauses written in implicational style. The proof search procedure for logic with only Horn clauses is well understood and widely used in many Prolog like logic programming languages. The higher-order types escape from the Horn clause fragment but they fit as a special case of first-order hereditary Harrop formulas.

The language of hereditary Harrop formulas is defined by a set of formulas G, whose validity can be automatically verified, and a set of formulas D which can be stated as axioms. The syntax for both is defined with two mutually dependent rules:

$$G ::= A \mid G \wedge G \mid G \vee G \mid \exists x.G \mid \forall x.G \mid D \rightarrow G$$
$$D ::= A \mid D \wedge D \mid \forall x.D \mid G \rightarrow A$$

where A stands for an atomic formula. The advantage of the family of hereditary Harrop formulas is that it is a non-trivial generalization of Horn clauses which permits a lot richer logical language but still allows efficient proof search. If we take the intersection of G and D, then we get the set F of all formulas that can be used both as goals and as axioms:

$$F ::= A \mid F \wedge F \mid \forall x.F \mid F \rightarrow A$$

Its implicational fragment:

$$F ::= A \mid \forall x.F \mid F \rightarrow F$$

is derived by eliminating the case for conjunction and also taking into account the currying transformation:

$$F_1 \wedge F_2 \rightarrow F_3 \equiv F_1 \rightarrow F_2 \rightarrow F_3$$

This fragment is exactly the set of GF types written in a style that is more traditional for classical logic. Namely we used the universal quantifier $\forall x$ instead of the dependent function type $(x : A) \rightarrow B$.

At least one Prolog variation - λProlog [Nadathur and Miller, 1988], supports the full language of hereditary Harrop formulas, and this is our source of inspiration for the proof search in GF. The theory and the implementation behind λProlog has been developed for more than twenty years and its most mature implementation Teyjus version 2 [Qi, 2009] provides a compiler and an interpreter which are efficient enough for doing non-trivial computations. Another dependently typed language with proof search inspired by λProlog is the language Twelf [Pfenning and Schürmann, 1999].

Besides the implementation of hereditary Harrop formulas, the other distinguishing feature of λProlog is that, unlike Prolog which supports only simple tree like data terms, λProlog permits arbitrary lambda terms. Since the abstract syntax trees are also arbitrary lambda terms, this is yet another reason to take it as a model for reasoning in GF. The cost of this generalization is that the simple first-order unification is no longer sufficient. Fortunately, the lesson from Miller [1991] tells us that for most practical applications full higher-order unification [Huet, 1975] is not needed and the weaker higher-order pattern unification is an efficient and still very useful alternative. The higher-order pattern unification is also used in Norell's type checker so it fits very well in a combination with the proof search.

In addition to the remarkable similarities between GF and λProlog, there are also some differences. First of all in GF we are not only interested in whether

a certain logical statement is true but we also want to have the exact proof term. In the context of λProlog, this would be overkill since the generation of proofs is equivalent to tracing the execution of the whole program which is far too expensive for a general purpose programming language. In GF, the typical abstract syntax trees rarely exceed the depth of 20-30 levels, and if we see the trees as proof terms, then they would correspond to rather small programs. In such cases, the overhead for constructing the proof term is acceptable and usually most of the time for search is spent on trying different alternatives.

Although both λProlog and GF operate with lambda terms they are not operationally identical. In GF, we permit definitional equalities for functions while in λProlog all functions symbols are just syntactic atoms and as such they act as data constructors from the type theoretic point of view. For instance in GF, we can define the addition of two Peano numbers as a function:

$$\begin{array}{l} \textbf{fun}\ \mathit{plus} : \mathit{Nat} \rightarrow \mathit{Nat} \rightarrow \mathit{Nat} \\ \textbf{def}\ \mathit{plus}\ \mathit{zero} \qquad y = y \\ \quad\ \ \mathit{plus}\ (\mathit{succ}\ x)\ y = \mathit{succ}\ (\mathit{plus}\ x\ y) \end{array}$$

while in λProlog it has to be implemented as a predicate with three arguments where the third argument represents the sum of the first two. The addition of function definitions changes the principal completeness of the high-order unification. For instance, if we have the unification problem:

$$?1\ (\mathit{succ}\ \mathit{zero})\ (\mathit{succ}\ (\mathit{succ}\ \mathit{zero})) \simeq \mathit{succ}\ (\mathit{succ}\ (\mathit{succ}\ \mathit{zero}))$$

where $?1$ is a metavariable, then in the absence of function definitions it has only the following three solutions:

$$\begin{array}{l} ?1 = \backslash x, y \rightarrow \mathit{succ}\ (\mathit{succ}\ (\mathit{succ}\ \mathit{zero})) \\ ?1 = \backslash x, y \rightarrow \mathit{succ}\ (\mathit{succ}\ x) \\ ?1 = \backslash x, y \rightarrow \mathit{succ}\ y \end{array}$$

However, when we take into account the function *plus*, then we get one more principal solution:

$$?1 = \backslash x, y \rightarrow \mathit{plus}\ x\ y$$

Unfortunately, finding the last solution is undecidable so instead we choose to leave the unification incomplete, i.e. we do not even try to find the last solution.

Actually, even the first two solutions can be computed only by using full higher-order unification which is far beyond the weaker higher-order pattern unification that we use. In that sense, adding function definitions does not make the pattern unification more incomplete than it is.

One advantage of the function definitions is that during the proof search the functional computations do not leave proof objects as foot prints and this makes them more efficient. As a general rule, it is a good practice to implement the deterministic computations as functions and the nondeterministic as logical rules.

Another difference is that while the λProlog programs are allowed to go into infinitely deep recursion, in GF the depth of the proof term is limited, i.e. we have a configuration option which constrains the upper limit of the number of recursive nestings. The rationale for this is that when the depth is limited, then the collection of generated trees is more representative. If we take as an example the Peano numbers and the function *plus*, then an exhaustive generation without depth limitation will start generating larger and larger numbers and will never backtrack and attempt to use the signature for *plus*. On the contrary, if we have an upper limit of three then the search will first generate the numbers from 0 to 2, and after that it will continue with terms like $\mathit{plus}\ 0\ 0$, $\mathit{plus}\ 0\ 1$, $\mathit{plus}\ 0\ 2$, $\mathit{plus}\ 1\ 0$, etc.

The depth limitation has also a positive effect on the search termination. Since there is only a finite number of terms with a given maximal depth, the search will always finish although it can take a long time. The termination guarantee, however, does not apply, if the search involves the computation of a function which does not terminate. The current GF compiler does not do any termination checking for functional definitions, and this is a major limitation which characterizes GF more as a dependently typed programming language and not as a sound theorem prover. In other words, for every theorem A we can define the function:

$$\begin{array}{l}\textbf{fun}\ \mathit{proof} : A \\ \textbf{def}\ \mathit{proof} = \mathit{proof}\end{array}$$

which is a fake proof for it. The termination checking can be added as a feature in the future versions of the compiler but even without it, GF is useful as a programming language.

Yet another extension to the search strategy is that while in λProlog the clauses for one and the same predicate are tried deterministically in a source code order, in GF we also support an alternative scheduling where the clauses are picked in a random order. More concretely, every abstract syntax function has a probability and every time when we have to refine an incomplete proof we take into account the specified probability distribution and we choose a function at random.

From the outline of the similarities and the differences between GF and the other systems, it is clear that many of the details in the implementation of our logical framework are not new. The difficulty, however, is that the relevant information is spread among many other sources and there is no single coherent reference. Moreover, even if we give a list of references, this is not enough since none of them is explicit in how the pieces fit together. In an attempt to overcome this shortcoming, and to make it easier for new developers and researchers, we decided that it will be beneficial to compile everything in a single detailed exposition. In the next few sections, we present the rules for computation, unification, type checking and proof search in the framework. The last section is devoted to the interaction between parsing, type checking and proof search which is the most GF specific part of this chapter.

3.1 Computation

The computation in the abstract syntax of GF is a simple generalization of the standard eval/apply strategy in lambda calculus. The two things that we have to add are metavariables and function definitions. Since we do the computation lazily, we need a representation for values, i.e. expressions reduced to head normal form:

Definition 9 *An abstract value is one of the following:*

- $(f\ as)$ *is an application where* as *is a list of argument values and* f *is either a data constructor or a function which requires more arguments than there are in* as*;*
- $(\underline{f}\ as)$ *is an application of an abstract function* f *which cannot be reduced further due to insufficient information;*
- $[\![\sigma, e]\!]$ *is a closure which closes the open expression* e *with the environment* σ *that gives values for the free variables in* e*. Here* σ *is a list of values where the* i*-th element in the list is the value for de Bruijn index* i*.*
- $(?i\ \sigma\ as)$ *is an application of the metavariable* $?i$ *where* as *is the list of arguments and* σ *is the environment that was in use when this value was computed;*

- $(!i\ \sigma\ as\ c)$ *represents a computation that is suspended until the metavariable* $?i$ *is substituted. The computation is resumed with the continuation c, i.e. with a function in the metalanguage that has to be executed;*
- $(@i\ as)$ *represents the application of a variable with de Bruijn level* i *to a list of arguments* as*;*

Here we should note that both in the expressions and in the values we have a representation for variables. The representation in the values is used when during the unification of two lambda terms, we have to do computation under lambda abstraction. In this case, we have unbound variables which in the value are represented with the notation $@i$. The difference, however, is that while in the expressions we use de Bruijn indices, i.e. $\#i$ points to the i-th binding counting from the variable reference to the left, in the values we have de Bruijn levels, i.e. $@i$ points to the i-th binding counting from the beginning of the expression to the right. The separation between expressions and values and the different representations for variables lets us do lazy substitution like in the suspension calculus [Nadathur and Wilson, 1994], without the need to renumber the Bruijn indices when the substitution is pushed under abstraction.

Another thing to point out is the representation of metavariables. While in Norell's treatment the metavariables can only be substituted with closed lambda terms, in our type checker we allow substitution with open terms. The difference is noticeable even with simple examples. If we have the expression:

$$\backslash x \rightarrow ?1$$

then we allow a substitution where $?1$ is directly instantiated with x which gives us the identity function:

$$\backslash x \rightarrow\ x$$

In Norell's typechecker this is not permitted since x is not a closed term. Instead the refinement proceeds in two steps. First the expression is rewritten to:

$$\backslash x \rightarrow ?1\ x$$

i.e. the metavariable is applied to all variables that are in the scope, and after that it is instantiated with a closed term which after β reduction will lead to the same result, i.e. we get from the type checker the term:

$$\backslash x \rightarrow\ (\backslash y \rightarrow y)\ x$$

which needs further simplifications. We avoid the generation of this more complicated expressions by representing the metavariables with values like $?i\ \sigma\ as$ where σ is an environment which closes the potential free variables in the substitution for $?i$. The environment basically represents the variables in the current scope so we get an equivalent result but we avoid the creation of redundant abstractions.

Another complication of the metavariables is that sometimes we have to suspend the computation until some metavariable is instantiated. For instance, if we have the expression:

$$\mathit{plus}\ ?1\ (\mathit{succ}\ \mathit{zero})$$

then we cannot compute it since we need to pattern match on the value of $?1$. Instead, we represent the computation with the suspension $!i\ \sigma\ as\ c$ which is resumed only when the metavariable $?i$ is instantiated. A suspended functional computation could on the other hand cause a suspension in the type checker and in the proof search. A suspended type checking problem is represented with a "guarded constant" in Norell's type checker while in the proof search algorithm this is a special case of dynamic rescheduling which Xiaochu Qi apply for hard unification problems.

Regarding the treatment of metavariables in the context of functional computations, there are two well know strategies in functional-logic languages [Hanus, 2006] – residuation and narrowing. In the residuation strategy, the computation is suspended until the variable is instantiated, while in the narrowing strategy, the variable is nondeterministically substituted with the possible values and the computation proceeds as normal. In Curry, the programmer can select the preferred strategy for every function while in GF we choose not to make the language more complicated and we fixed the strategy to residuation. The narrowing strategy can be emulated by using logical predicates instead of functions so this is not a major restriction.

In the definitions above and later, we use lists of values for which we introduce the following notations:

- *nil* is an empty list;
- $v :: a$ is a list with head the value v and tail a;
- $a[i]$ is the i-th element of the list.
- $a +\!\!+\ b$ is the concatenation of the lists a and b

Having set the background details, we can now define the computational rules with Haskell like pseudocode. The main part of the computation is done in the functions *eval* and *apply*. The reduction of an expression to a value is done in the function *eval*:

$$
\begin{array}{lll}
\textit{eval}\ \sigma\ (\#i) & = \sigma[i] & \\
\textit{eval}\ \sigma\ (f) & = \textit{match}\ f\ \textit{eqs}\ \textit{nil} & \text{, if } f \text{ is a function defined with } eqs \\
\textit{eval}\ \sigma\ (e_1\ e_2) & = \textit{apply}\ \sigma\ e_1\ ((\textit{eval}\ \sigma\ e_2) :: \textit{nil}) & \\
\textit{eval}\ \sigma\ (e_1\ \{e_2\}) & = \textit{apply}\ \sigma\ e_1\ ((\textit{eval}\ \sigma\ e_2) :: \textit{nil}) & \\
\textit{eval}\ \sigma\ (\backslash x \rightarrow e) & = [\![\sigma, \backslash x \rightarrow e]\!] & \\
\textit{eval}\ \sigma\ (\backslash\{x\} \rightarrow e) & = [\![\sigma, \backslash\{x\} \rightarrow e]\!] & \\
\textit{eval}\ \sigma\ (?i) & = ?i\ \sigma\ \textit{nil} &
\end{array}
$$

When *eval* has to evaluate an application, then it calls *apply*, which on the other hand calls back to *eval* for the evaluation of the function arguments:

$$
\begin{array}{llll}
\textit{apply}\ \sigma\ e & \textit{nil} & = \textit{eval}\ \sigma\ e & \\
\textit{apply}\ \sigma\ (\#i) & as & = \textit{apply'}\ \sigma[i]\ as & \\
\textit{apply}\ \sigma\ (f) & as & = \textit{match}\ f\ as & \text{, if } f \text{ is a function defined with } eqs \\
\textit{apply}\ \sigma\ (e_1\ e_2) & as & = \textit{apply}\ \sigma\ e_1\ (\textit{eval}\ \sigma\ e_2 :: as) & \\
\textit{apply}\ \sigma\ (e_1\ \{e_2\}) & as & = \textit{apply}\ \sigma\ e_1\ (\textit{eval}\ \sigma\ e_2 :: as) & \\
\textit{apply}\ \sigma\ (\backslash x \rightarrow e) & (v :: as) & = \textit{apply}\ (v :: \sigma)\ e\ as & \\
\textit{apply}\ \sigma\ (\backslash\{x\} \rightarrow e) & (v :: as) & = \textit{apply}\ (v :: \sigma)\ e\ as & \\
\textit{apply}\ \sigma\ (?i) & as & = ?i\ \sigma\ as &
\end{array}
$$

Note that the pseudocode is written in such a way that the evaluation strategy, i.e. strict vs lazy evaluation, depends on the evaluation strategy of the host language. Whenever there is an application, i.e. an expression like $(e_1\ e_2)$, then the value of e_2 is pushed into the stack of arguments by the code $\textit{eval}\ \sigma\ e_2 :: as$. Since in the current implementation the host language is Haskell, $\textit{eval}\ \sigma\ e_2$ will not be evaluated until needed, and this is enough to realize lazy evaluation. In strict languages like ML, the implementation will need some extra bookkeeping in order to ensure lazy evaluation.

When either *eval* or *apply* needs to evaluate some abstract function f, then the list of definitional equations eqs is retrieved and the matching is done in the helper

function *match*:

$$
\begin{array}{ll}
\mathit{match}\ f\ \mathit{nil} & as_0 = \underline{f}\ as_0 \\
\mathit{match}\ f\ ((ps,e) :: eqs) & as_0 = \mathit{tryMatches}\ ps\ as_0\ e\ \mathit{nil}
\end{array}
$$

where

$$
\begin{array}{lll}
\mathit{tryMatches}\ \mathit{nil} & as & e\ \sigma = \mathit{apply}\ \sigma\ e\ as \\
\mathit{tryMatches}\ (p :: ps) & (a :: as) & e\ \sigma = \mathit{tryMatch}\ p\ a\ \sigma
\end{array}
$$

where

$$
\begin{array}{lll}
\mathit{tryMatch}\ x & a & \sigma = \mathit{tryMatches}\ ps\ as\ e\ (a :: \sigma) \quad \text{, if } x \text{ is a variable} \\
\mathit{tryMatch}\ p & (?i\ \sigma_i\ as') & \sigma = !i\ \sigma_i\ as'\ (\backslash v \rightarrow \mathit{tryMatch}\ p\ v\ \sigma) \\
\mathit{tryMatch}\ p & (!i\ \sigma_i\ as'\ c) & \sigma = !i\ \sigma_i\ as'\ (\backslash v \rightarrow \mathit{tryMatch}\ p\ (c\ v)\ \sigma) \\
\mathit{tryMatch}\ p & (@i\ as') & \sigma = \underline{f}\ as_0 \\
\mathit{tryMatch}\ p & (\underline{g}\ as') & \sigma = \underline{f}\ as_0 \\
\mathit{tryMatch}\ (g\ ps') & (\underline{g}\ as') & \sigma = \mathit{tryMatches}\ (ps' +\!\!+\ ps)\ (as' +\!\!+\ as)\ e\ \sigma \\
\mathit{tryMatch}\ _ & _ & \sigma = \mathit{match}\ f\ eqs\ as_0
\end{array}
$$

The function iterates over the list of equations and tries sequentially the patterns until a match is found. Every equation is represented as a pair (ps, e) of a list of patterns ps and an expression e. The pattern is either a variable which binds to the value of the argument or a constructor pattern like $f\ ps$ where f is the constructor and ps is a list of new patterns. The expression in the equation may contain free variables which are bound by the variable patterns in ps.

The pattern matching is quite straightforward and there are only two things that are worth emphasizing. First of all this is the place where we have to implement the residuation strategy. If the current pattern is a variable, then any kind of argument value is acceptable, and we can continue with the matching of the rest of the patterns. If the pattern is not a variable then the only other choice is to have a constructor pattern, but before we do the matching we have to be sure that the value is not a metavariable. For this reason the second line for *tryMatch* check for a metavariable, and if it is, then the computation is immediately suspended by returning a suspension value. Similarly, if we pattern match on an already created suspension then we create another suspension whose continuation calls the continuation from the original suspension.

The other interesting case is when the value of the argument is a variable, i.e. a value of the kind $@i$. We cannot pattern match on variables, and furthermore, it does not make sense to create suspensions in this case because contrary to the metavariables the usual variables are bound by a lambda abstraction or a dependent type, so they will not get definite values in any situation. From the logical

point of view the variables are universally quantified while the metavariables are existentially quantified. When the application of a function f depends on the value of a universally quantified variable, then we have no other choice, except to keep the whole expression in a partially evaluated form. For this reason in *tryMatch* we return the value $\underline{f}\ as_0$ which is just the original function f applied to the already evaluated arguments. However we use $\underline{f}\ as_0$ instead of $f\ as_0$ because the former tells us that no further reduction is possible, so we can avoid unnecessary computations. The fifth line in *tryMatch* is exactly this, it checks whether the value of the argument is a partially evaluated expression and if it is, it returns another partial value.

The value $\underline{f}\ as_0$ is returned also in the case when the patterns in the definition of f are incomplete and we encounter a case which is not covered by any of the equations. This situation is handled by the first line in the definition of *match* and the resultant behaviour is rather different from the traditional functional languages like Haskell and ML, where an incomplete pattern matching leads to a runtime error. The reason is that while in the functional languages there is no computational use for partially defined function, in GF such functions can still be linearized and thus can be useful in natural language generation tasks where the function definitions are used as a way to generate paraphrases.

If the value of the argument is neither of the discussed kinds then it is safe to compare the constructors in the constructor pattern and in the value. If there is a match, the new patterns are added to the existing ones and the matching continues. If the matching fails, then we restart the matching with the next equation in the definition of f.

Finally we must add one more helper function *apply'* which is called by *apply* when the expression is a variable $\#i$. This happens when we have variables of a function type and in this case we have to apply already computed value to more arguments. The definition of *apply'* simply adds more arguments:

$$
\begin{array}{lll}
\textit{apply'}\ v & \textit{nil} & = v \\
\textit{apply'}\ (f\ vs_0) & vs & = \textit{apply nil}\ f\ (vs_0 +\!\!+\ vs) \\
\textit{apply'}\ (?i\ \sigma\ vs_0) & vs & = ?i\ \sigma\ (vs_0 +\!\!+\ vs) \\
\textit{apply'}\ (@i\ vs_0) & vs & = @i\ (vs_0 +\!\!+\ vs) \\
\textit{apply'}\ (!i\sigma\ vs_0\ k) & vs & = !i\ \sigma\ vs_0\ (\backslash v \rightarrow \textit{apply'}\ (k\ v)\ vs) \\
\textit{apply'}\ (\underline{f}\ vs_0) & vs & = \underline{f}\ (vs0 +\!\!+\ vs) \\
\textit{apply'}\ [\![\sigma, \backslash x \rightarrow e]\!] & (v :: vs) & = \textit{apply}\ (v :: \sigma)\ e\ vs
\end{array}
$$

3.2 Higher-order Pattern Unification

The aim of the unification algorithm is to compare two values, and if it is possible, to produce a substitution for the metavariables that equalizes the two values. The complication is that it is completely legal to use metavariables as functions. For instance consider again the unification problem:

$$?1\ (\mathit{succ}\ \mathit{zero})\ (\mathit{succ}\ (\mathit{succ}\ \mathit{zero})) \simeq \mathit{succ}\ (\mathit{succ}\ (\mathit{succ}\ \mathit{zero}))$$

i.e. we want to unify the value $(?1\ (\mathit{succ}\ \mathit{zero})\ (\mathit{succ}\ (\mathit{succ}\ \mathit{zero})))$ with the value $(\mathit{succ}\ (\mathit{succ}\ (\mathit{succ}\ \mathit{zero})))$. This is undecidable since every function which maps the arguments $(\mathit{succ}\ \ \mathit{zero})$ and $(\mathit{succ}\ \ (\mathit{succ}\ \ \mathit{zero}))$ into $(\mathit{succ}\ (\mathit{succ}\ (\mathit{succ}\ \mathit{zero})))$ is a valid solution. The problem is avoided by restricting the unification from general higher-order unification to the limited higher-order pattern unification. The later has the advantage that it is decidable and efficient, and furthermore it is guaranteed that there is always at most one most general substitution. The disadvantage of the higher-order pattern unification is that it cannot solve all unification problems. For instance the problem above is not solvable because the possible substitution for $?1$ is not uniquely determined. We call such problems – hard unification problems, and instead of trying to solve them directly, we postpone the resolution until there is some more information which simplifies the problem, i.e. until the problematic metavariable is substituted in some other branch of the overall processing.

In order to explain better the idea behind higher-order pattern unification, we must introduce the notion of a scope for a metavariable. The general definition of scope is the list of typed variables that are available at a given spot of an expression, i.e. a list of $x : [\![\sigma, A]\!]$ where x is a variable name and $[\![\sigma, A]\!]$ is its type represented as a closure of the actual type A with the environment σ which assigns values to the free variables in A. Here the variable that is bound by the innermost lambda abstraction is the first in the list. Since in the original expression, every metavariable appears only once, it naturally gets associated with the scope that is active at the spot where the metavariable appeared. When an expression is computed or typechecked it can happen that some of its metavariables can get multiplied but still in this case we can assign to such metavariable the part of the scope that is available in all appearances of the metavariable.

Now the limitation of the higher-order pattern unification is that it can handle only problems where if an unbound metavariable is used as a function, then it can only be applied to a distinct list of variables that are not in the scope of the

metavariable. For instance, we can solve the unification problem:

$$?1\ x \simeq e$$

where e is some expression and x is a variable that is not in the scope of $?1$. The only solution in this case is to substitute $\backslash x \rightarrow e$ for $?1$. Any other solution for instance $\backslash y \rightarrow x$ is invalid since this will let the variable x to escape from its scope. In order to guarantee the validity of the substitution we must also perform occurs check which guarantees that all variables in e are either in the scope of $?1$ or are given as arguments in the application of $?1$. Since we also require that the arguments are distinct variables, this guarantees that there is a unique solution for the substitution. Furthermore, if the expression e contains other metavariables, then the occurs check must ensure that they have scope that is more shallow than the scope of $?1$. This guarantees that further substitutions will not let some variables to escape out of their scope. Note that this also prevents the construction of infinite terms since $?1$ is not allowed to appear in e.

Formally, the algorithm for unification of two values v_1 and v_2 is represented as a set of deduction rules for the unifiability statement:

$$k, j \vdash v_1 \simeq v_2$$

Here the parameter k in the statement is used for determining the de Bruijn levels for the variables when we compute under lambda, and the parameter j is the index of the guarded constant which will be used if we are faced with hard unification problems. For instance, the type checker might determine that it is not possible to certify whether some expression e is type correct until some metavariable $?i$ is instantiated. The solution for this is that the type checker creates another metavariable (guarded constant) $?j$ which will represent e in the further computations. The name of the guarded constant is passed to the unification algorithm, and if it is not able to solve the current problem, then it locks the binding $?j = e$ until it is possible to resume the unification, i.e. until $?i$ is instantiated. Similar situation arises in the proof search where it might not be possible to determine whether certain refinement preserves the correctness of the partial proof.

In the unifiability statement, we left the generated substitution implicit since otherwise the deduction rules will become too verbose. Alternatively, we could have used statement like:

$$\theta_1; k, j \vdash v_1 \simeq v_2 \rightsquigarrow \theta_2$$

where θ_1 and θ_2 are the substitutions before and after the unification, but then the explicit passing of the updated substitution will make the deduction rules too heavy. We choose to make the flow implicit by assuming that the substitution is always passed from left to right and from top to down. The state of every metavariable in the substitution can be one of:

- **unbound** - a metavariable that still has not been assigned. This state is characterized with the triple $(\Gamma, [\![\sigma, A]\!], \mathit{cs})$ where Γ is the scope of the metavariable, $[\![\sigma, A]\!]$ is its type and *cs* is a list of constraints, i.e. delayed unification problems, that have to be evaluated after the variable is substituted;

- **bound** - the variable has been substituted with some expression e. This state is characterized with the pair (Γ, e) where e is a potentially open expression whose free variables are defined in the scope Γ;

- **guarded** - this state corresponds to the concept of guarded constants. The state is characterized by the triple (e, cs, n) where again e is an expression, cs a list of constraints, and n is a lock counter, i.e. a natural number indicating the number of metavariables that have to be substituted in order to unlock the currently guarded metavariable.

The only way to change the state of some metavariable is by calling one of these primitive operations:

- $i \leftarrow$ *newMeta* Γ $[\![\sigma, A]\!]$ - introduces a new metavariable whose index is i and remembers the current scope Γ and its type $[\![\sigma, A]\!]$. The new variable has an empty list of constraints.

- $j \leftarrow$ *newGuard* e - introduces a new guarded constant with index j which guards the expression e. The lock counter is zero and the list of constraints is empty.

- *addConstraint* j i c - locks the guarded constant j and attaches the constraint c to the metavariable i. When $?i$ is substituted then this will unlock j, i.e. it will decrease the lock counter for j.

- *setMeta* i e - binds the metavariable i with the expression e if the expression satisfies the occurs check condition, and fails otherwise.

Furthermore, when it is necessary, the unification algorithm calls the algorithms *eval* and *apply* for abstract computation. Since now the metavariables can be

substituted, the substitution has to be made transparent during the computation. In other words, when the computation algorithm encounters a metavariable, then it first checks whether the metavariable is already substituted, and if it is, then the computation proceeds with the substituted value. If it is not substituted, then the rules for computation with metavariables are used. In the previous chapter, we left out the details for handling the substitution since this would make the pseudocode less readable.

The the unification rules are listed on Figure 3.1. The first two rules correspond to the case when one of the unified values represents a computation that is suspended due to some unbound variable $?i$. In this case, the unification process is also suspended by calling *addConstraint* and asserting the constraint that if we resume the evaluation with the substitution for $?i$, then we must get a value unifiable with the other given value.

The next three rules handle the unification of values that are headed by a metavariable. In the first case, the metavariable is unified with itself, which is allowed only if the two values carry the same environments and the same list of arguments, i.e. we have to unify the lists $(\sigma_1 +\!\!+\ as_1)$ and $(\sigma_2 +\!\!+\ as_2)$. If a metavariable is unified with another value, then it must be bound with the given value. The binding itself is handled with the predicate *bind*:

bind $k\ j\ i\ \sigma\ as\ v =$
  if $(\sigma +\!\!+\ as)$ are distinct variables and not in the scope of $?i$
    then *setMeta* $i\ (\backslash x_1, \ldots x_{|as|} \rightarrow v)$
    else *addConstraint* $j\ i\ (\backslash e \rightarrow k, j \vdash$ *apply* $\sigma\ e\ as \simeq v)$

It checks the condition that the arguments and the environment of the metavariable are constituted of only distinct variables. This is the precondition that has to be satisfied for the pattern matching unification to be successful. If this condition is not satisfied, then we have encountered a hard unification problem that cannot be resolved. This is the other case when the whole unification process must be suspended until the metavariable is substituted by resolving some other goals in the type checker or in the proof search. The constraint that is attached to the metavariable will resume the current unification after the metavariable is bound.

Three more rules handle the cases when the values are either function applications ($f\ as$), irreducible applications ($\underline{f}\ as$), or variables ($@i\ as$). In this cases the unification is reduced to the unification of the corresponding arguments as.

Finally, we must handle the unification of lambda abstractions. If both values are lambda abstractions, then we evaluate the expressions under the lambdas with

$$\frac{\textit{addConstraint}\ j\ i\ (\backslash e \rightarrow k, j \vdash c\ (\textit{apply}\ \sigma_1\ e\ as_1) \simeq v_2)}{k, j \vdash\ !i\ \sigma_1\ as_1\ c \simeq v_2}$$

$$\frac{\textit{addConstraint}\ j\ i\ (\backslash e \rightarrow k, j \vdash c\ (\textit{apply}\ \sigma_1\ e\ as_1) \simeq v_2)}{k, j \vdash v_1 \simeq\ !i\ \sigma_2\ as_2\ c}$$

$$\frac{\forall l. \quad k, j \vdash (\sigma_1 +\!\!+\ as_1)[l] \simeq (\sigma_2 +\!\!+\ as_2)[l]}{k, j \vdash (?i\ \sigma_1\ as_1) \simeq (?i\ \sigma_2\ as_2)}$$

$$\frac{\textit{bind}\ k\ j\ i\ \sigma_1\ as_1\ v_2}{k, j \vdash (?i\ \sigma_1\ as_1) \simeq v_2} \qquad \frac{\textit{bind}\ k\ j\ i\ \sigma_2\ as_2\ v_1}{k, j \vdash v_1 \simeq (?i\ \sigma_2\ as_2)}$$

$$\frac{\forall l. \quad k, j \vdash as_1[l] \simeq as_2[l]}{k, j \vdash f\ as_1 \simeq f\ as_2} \qquad \frac{\forall l. \quad k, j \vdash as_1[l] \simeq as_2[l]}{k, j \vdash \underline{f}\ as_1 \simeq \underline{f}\ as_2}$$

$$\frac{\forall l. \quad k, j \vdash as_1[l] \simeq as_2[l]}{k, j \vdash @i\ as_1 \simeq @i\ as_2}$$

$$\frac{(k+1), j \vdash \textit{eval}\ (@k\ \textit{nil} :: \sigma_1)\ e_1 \simeq \textit{eval}\ (@k\ \textit{nil} :: \sigma_2)\ e_2}{k, j \vdash [\![\sigma_1, \backslash x_1 \rightarrow e_1]\!] \simeq [\![\sigma_2, \backslash x_2 \rightarrow e_2]\!]}$$

$$\frac{(k+1), j \vdash \textit{eval}\ (@k\ \textit{nil} :: \sigma_1)\ e_1 \simeq \textit{apply'}\ v_2\ (@k\ \textit{nil})}{k, j \vdash [\![\sigma_1, \backslash x_1 \rightarrow e_1]\!] \simeq v_2}$$

$$\frac{(k+1), j \vdash \textit{apply'}\ v_1\ (@k\ \textit{nil}) \simeq \textit{eval}\ (@k\ \textit{nil} :: \sigma_2)\ e_2}{k, j \vdash v_1 \simeq [\![\sigma_2, \backslash x_2 \rightarrow e_2]\!]}$$

Figure 3.1: Unification of Values

$$\frac{k, j \vdash [\![\sigma_1, A_1]\!] \simeq [\![\sigma_2, A_2]\!] \qquad (k+1), j \vdash [\![@k\ \textit{nil} :: \sigma_1, B_1]\!] \simeq [\![@k\ \textit{nil} :: \sigma_2, B_2]\!]}{k, j \vdash [\![\sigma_1, (x_1 : A_1) \rightarrow B_1]\!] \simeq [\![\sigma_2, (x_2 : A_2) \rightarrow B_2]\!]}$$

$$\frac{k, j \vdash [\![\sigma_1, A_1]\!] \simeq [\![\sigma_2, A_2]\!] \qquad (k+1), j \vdash [\![@k\ \textit{nil} :: \sigma_1, B_1]\!] \simeq [\![@k\ \textit{nil} :: \sigma_2, B_2]\!]}{k, j \vdash [\![\sigma_1, (\{x_1\} : A_1) \rightarrow B_1]\!] \simeq [\![\sigma_2, (\{x_2\} : A_2) \rightarrow B_2]\!]}$$

$$\frac{k, j \vdash [\![\sigma_1, A_1]\!] \simeq [\![\sigma_2, A_2]\!] \qquad k, j \vdash [\![\sigma_1, B_1]\!] \simeq [\![\sigma_2, B_2]\!]}{k, j \vdash [\![\sigma_1, A_1 \rightarrow B_1]\!] \simeq [\![\sigma_2, A_2 \rightarrow B_2]\!]}$$

$$\frac{\forall i. \quad k, j \vdash \textit{eval}\ \sigma_1\ e_{1i} \simeq \textit{eval}\ \sigma_2\ e_{2i}}{k, j \vdash [\![\sigma_1, C\ e_{11} \ldots e_{1n}]\!] \simeq [\![\sigma_2, C\ e_{21} \ldots e_{2n}]\!]}$$

Figure 3.2: Unification of Types

an environment that is extended with the value $@k$ *nil*. After that we unify the computed values. Basically this performs partial evaluation since the value $@k$ indicates that we are dealing with something that is not known at this time, i.e. it represents the variable bound by the abstraction. Despite that both values must have the same type, i.e. they are both functions, it is not necessary that they both are lambda abstractions. For example, one of the values can be produced by partial application while the other is a full lambda abstraction. In this case we perform on-the-fly raising, i.e. we directly apply the partial application to the value $@k$. This raises the partial application to full lambda abstraction.

In addition to the unification of values, in the type checker, we will also need to unify types. In the type unification we basically check that the two types have the same shape and that the values of all possible dependencies are also unifiable. The rules for type unification are in Figure 3.2. The first three rules implement structural induction over the structure of the types where we must remember to extend the environment of each type, if it is a dependent function type. The last rule handles the basic case where we have the application of some category C over a list of expressions. In this case we evaluate the expressions in the environments of the types and unify the computed values.

3.3 Type Checking

In its simplest form, type checking is a process which starts from some abstract expression and a given type, and it verifies whether the expression has the same type. In the presence of dependent types, metavariables and implicit arguments, however, the process is more complicated. It might be the case that the given expression has the same type only under the assumption that certain metavariables are instantiated with the right values. In this case, the type checker should also compute the substitution. Furthermore, if the expression has hidden implicit arguments, then they must be made explicit. Since the discovery of implicit arguments is directed by the types of the subexpressions, this must be done simultaneously with the type checking. The consequence is that the type checker not only verifies the type but it also returns a new refined expression where some metavariables might be instantiated and all implicit arguments are made explicit. Finally, we said that we use de Bruijn representation for variables but actually it is not possible to assign the indices in advance, since the type checker might have to introduce implicit lambda abstractions which will require complex renumbering of the de Bruijn indices. For instance, if we naively assign the indices in the expression:

$$(\backslash x \rightarrow x) : (x, \{y\} : A) \rightarrow B$$

in advance, then we should assign $\#0$ for x. The type of the expression, however, suggests that it must be expanded to:

$$(\backslash x, \{y\} \rightarrow x) : (x, \{y\} : A) \rightarrow B$$

where x now has index $\#1$. Instead of recomputing the indices in the type checker, we assume that the variables and the function names in the source expression are indistinguishable and are both represented as names. The type checker maintains the list of all variable names in the current scope, and thus it is able to decide which names refer to local variables and which to global constants defined in the grammar. In the refined expression, the type checker generates de Bruijn index, if the current name is a variable and retains the name, if it refers to a global constant. However, the unification and the evaluation algorithms will never see plain variable names since they are already replaced in all expressions that the type checker needs to compute or unify.

For most of the expressions, it is possible to directly match the form of the expression with its type. In other cases, it is easier to infer the type from the expression, and after that the inferred typed is unified with the type given to the type

checker. For this reason there are two main procedures - type checking and type inference. In addition, there is a third procedure which inserts implicit arguments when it is necessary. We define all of them as deduction rules for three different statements:

1. type checking - $\Gamma \vdash e \uparrow [\![\sigma, B]\!] \rightsquigarrow e'$
2. type inference - $\Gamma \vdash e \downarrow [\![\sigma, B]\!] \rightsquigarrow e'$
3. implicit arguments - $\Gamma \vdash e : [\![\sigma, B]\!] \rightsquigarrow e' : [\![\sigma', B']\!]$

Here Γ is the current scope, e is the source expression and e' is the refined expression that we compute as result. The difference between the first two statements is that while in the type checking we know the type $[\![\sigma, B]\!]$, in the type inference we must infer it. The rules for type checking, type inference and arguments insertion are given on Figures 3.3, 3.4 and 3.5.

The first and the third rule in the type checker are for checking lambda abstractions with dependent function types. The only difference is that in the first case the abstraction is with an implicit argument while in the second case it is explicit. In both cases, we add the variable x with type $[\![\sigma, A]\!]$ to the scope Γ, and we type check the body of the abstraction against the type B. Since B is under binding, we must add a variable in the environment for B to account for the free occurrence of y in B. The de Bruijn level is equal to the size of the scope $|\Gamma|$ since the value for y must correspond to the value for x and x has just been added to the scope.

The second rule accounts for the case when an expression which is not an implicit lambda abstraction is checked against an implicit function type. The rule is exactly the same as the first one except that we must generate a fresh variable x, and in the output expression, we must insert the implicit abstraction that was omitted in the source expression.

The fourth rule is for type checking with simple function types. The processing is the same except that we do not have to extend the environment for B.

The metavariables in the source expression do not have explicit indices, and instead, the indices are assigned by the type checker. The type checking of the metavariables is implemented with the next rule. Here we just create a new metavariable and remember the current scope and type. In the output expression, we put the newly generated index.

The last rule is the most complicated and it covers the cases in which the type cannot be checked directly. In this case, the type checking procedure calls

$$\frac{(x : [\![\sigma, A]\!]) :: \Gamma \vdash e \uparrow [\![(@|\Gamma|\ \textit{nil}) :: \sigma, B]\!] \rightsquigarrow e'}{\Gamma \vdash (\backslash\{x\} \to e) \uparrow [\![\sigma, (\{y\} : A) \to B]\!] \rightsquigarrow (\backslash\{x\} \to e')}$$

$$\frac{(x : [\![\sigma, A]\!]) :: \Gamma \vdash e \uparrow [\![(@|\Gamma|\ \textit{nil}) :: \sigma, B]\!] \rightsquigarrow e'}{\Gamma \vdash e \uparrow [\![\sigma, (\{y\} : A) \to B]\!] \rightsquigarrow (\backslash\{x\} \to e')} \qquad e \neq (\backslash\{x\} \to e'')$$

$$\frac{(x : [\![\sigma, A]\!]) :: \Gamma \vdash e \uparrow [\![(@|\Gamma|\ \textit{nil}) :: \sigma, B]\!] \rightsquigarrow e'}{\Gamma \vdash \backslash x \to e \uparrow [\![\sigma, (y : A) \to B]\!] \rightsquigarrow (\backslash x \to e')}$$

$$\frac{(x : [\![\sigma, A]\!]) :: \Gamma \vdash e \uparrow [\![\sigma, B]\!] \rightsquigarrow e'}{\Gamma \vdash \backslash x \to e \uparrow [\![\sigma, A \to B]\!] \rightsquigarrow (\backslash x \to e')}$$

$$\frac{i \leftarrow \textit{newMeta}\ \Gamma\ [\![\sigma, A]\!]}{\Gamma \vdash ? \uparrow [\![\sigma, A]\!] \rightsquigarrow ?i}$$

$$\frac{\begin{array}{ll}\Gamma \vdash e \downarrow [\![\sigma'', A'']\!] \rightsquigarrow e'' & \Gamma \vdash e'' : [\![\sigma'', A'']\!] \rightsquigarrow e' : [\![\sigma', A']\!] \\ i \leftarrow \textit{newGuard}\ e' & |\Gamma|,\ i\ \vdash\ [\![\sigma, A]\!] \simeq [\![\sigma', A']\!]\end{array}}{\Gamma \vdash e \uparrow [\![\sigma, A]\!] \rightsquigarrow ?i}$$

Figure 3.3: Type Checking Rules

$$\frac{\Gamma \vdash e_1 \downarrow [\![\sigma'', C]\!] \rightsquigarrow e_1'' \qquad \Gamma \vdash e_1'' : [\![\sigma'', C]\!] \rightsquigarrow e_1' : [\![\sigma, (y : A) \to B]\!] \qquad \Gamma \vdash e_2 \uparrow [\![\sigma, A]\!] \rightsquigarrow e_2'}{\Gamma \vdash e_1\ e_2 \downarrow [\![\mathit{eval}\ \sigma_\Gamma\ e_2' :: \sigma, B]\!] \rightsquigarrow e_1'\ e_2'}$$

$$\frac{\Gamma \vdash e_1 \downarrow [\![\sigma'', C]\!] \rightsquigarrow e_1'' \qquad \Gamma \vdash e_1'' : [\![\sigma'', C]\!] \rightsquigarrow e_1' : [\![\sigma, A \to B]\!] \qquad \Gamma \vdash e_2 \uparrow [\![\sigma, A]\!] \rightsquigarrow e_2'}{\Gamma \vdash e_1\ e_2 \downarrow [\![\sigma, B]\!] \rightsquigarrow e_1'\ e_2'}$$

$$\frac{\Gamma \vdash e_1 \downarrow [\![\sigma, (\{y\} : A) \to B]\!] \rightsquigarrow e_1' \qquad \Gamma \vdash e_2 \downarrow [\![\sigma, A]\!] \rightsquigarrow e_2'}{\Gamma \vdash e_1\ \{e_2\} \downarrow [\![\mathit{eval}\ \sigma_\Gamma\ e_2' :: \sigma, B]\!] \rightsquigarrow e_1'\ \{e_2'\}}$$

$$\frac{}{\Gamma \vdash x \downarrow [\![\sigma, A]\!] \rightsquigarrow \#i} \quad \Gamma[i] \equiv (x : [\![\sigma, A]\!])$$

$$\frac{}{\Gamma \vdash c \downarrow [\![\mathit{nil}, A]\!] \rightsquigarrow c} \quad c : A \text{ is defined in the grammar}$$

Figure 3.4: Type Inference Rules

the type inference which infers the type $[\![\sigma'', A'']\!]$ and the refined expression e''. Since A'' might be an implicit function type, i.e. $(\{x\} : B) \to C$, we must call the procedure for insertion of implicit arguments which from $[\![\sigma'', A'']\!]$ and e'' derives a new expression e', where all potential implicit arguments are inserted, and its type is $[\![\sigma', A']\!]$. Finally, $[\![\sigma', A']\!]$ is unified with the type $[\![\sigma, A]\!]$ against which we need to check. Since the unification might not be solvable, we also create a metavariable i which guards the already refined expression e'. The index i is passed to the unification procedure which will lock the metavariable, if the problem is not solvable.

The first and the third inference rules are for inference from implicit and explicit applications (Figure 3.4). If the application is explicit, we first infer the type of e_1 and we get its type $[\![\sigma'', C]\!]$ and the refined expression e_1''. Now since some implicit arguments might be omitted in the source expression, we again run the procedure for insertion of arguments. It produces some new expression e_1' where the arguments are inserted and its type which now must have the shape $[\![\sigma, (y : A) \to B]\!]$ for the type checking to be successful. If this is the case, then we simply typecheck the argument e_2 against the type $[\![\sigma, A]\!]$. If the application is

$$\frac{i \leftarrow \textit{newMeta}\ \Gamma\ [\![\sigma, A]\!] \qquad \Gamma \vdash e\ \{?i\} : [\![?i\ \sigma\ \textit{nil} :: \sigma, B]\!] \leadsto e' : [\![\sigma', C]\!]}{\Gamma \vdash e : [\![\sigma, (\{x\} : A) \rightarrow B]\!] \leadsto e' : [\![\sigma', C]\!]}$$

$$\frac{}{\Gamma \vdash e : [\![\sigma, C]\!] \leadsto e : [\![\sigma, C]\!]} \qquad C \neq (\{x\} : A) \rightarrow B$$

Figure 3.5: Rules for Insertion of Implicit Arguments

implicit, then the processing is similar except that we do not do argument insertion and the inferred type for e_1 must be $[\![\sigma, (\{y\} : A) \rightarrow B]\!]$. Note that in both cases the inferred type for the whole expression is equal to B but in an environment that is extended with the value of argument e'_2. Here e'_2 is computed in the environment σ_Γ which is obtained from the scope Γ by replacing each variable in the scope with the value $(@i\ \textit{nil})$ where i is the de Bruijn level of the variable in the scope.

The second rule is a variation of the first rule where the type of e'_1 happens to be a non dependent function. Otherwise, the only difference with the first one is that we do not extend the environment for B.

The last two inference rules are very similar, and they cover the case where the source expression is either a local variable or a global function defined in the grammar. As we said, they are both represented by name in the source expression, but the type checker separates the two cases by looking up the name in the current scope. If the name is for local variable, then it is rewritten as a de Bruijn index, equal to the position at which the variable was found in the scope. Otherwise the type is inferred from the definition of the global function in the grammar.

Finally the insertion of implicit arguments is guided by two rules shown on Figure 3.5. The basic idea is that if the type of the expression e is an implicit function $(\{x\} : A) \rightarrow B$, then we apply the expression to an implicit argument which is just a metavariable, i.e. we get $e\ \{?i\}$. After that we repeat the procedure with the new expression until its type is no longer an implicit function.

3.4 Proof Search

The proof search starts either from the type of a target expression or from an incomplete expression where the incomplete parts are filled in with metavariables. In the second case, the expression is first passed to the type checker which explicitly stores the type of every metavariable. After that each metavariable and its type are seen as a new goal which has to be satisfied.

The actual search procedure is an implementation of connection tableaux (Andrews [1981], Bibel [1982]) with two simple extensions. The first is that the terms in the language are arbitrary lambda expressions, and the second is that we permit goals of function type which involves hypothetical reasoning. We get for free the handling for lambda expressions, since we already have an algorithm for high-order pattern unification in the type checker. When we have, as a goal, a function type like $A \to B$, then a pure first-order reasoner would convert it to a classically equivalent clause, i.e. $\neg A \vee B$. This equivalence, however, does not hold in the intuitionistic logic, and we should treat the implications directly. If we have to prove that $A \to B$ is true, then we temporarily assume the hypothesis A and we try to prove B in the new context. If the proof search is successful, then we have the proof $e : B$, from which we can construct the final result $\backslash x \to e : A \to B$. This is exactly the strategy that is adopted in λProlog too. During the proof search, the interpreter keeps track of the currently active hypotheses, and every time when some goal has to be resolved, both the globally defined clauses and the local hypotheses are considered. From the type theoretic point of view, the list of hypotheses is nothing else but just a scope with local variables.

During the search we need a representation for partial proofs, and the metavariables that we introduced in the previous sections are a convenient tool for both a representation of partial proofs and as a guidance for the search direction. At every step we maintain a storage which contains the state of every metavariable, either bound or not. If the search is initiated by the target type alone, then the initial storage contains a single unbound variable with the designated type. When the search starts from some incomplete term, then the storage contains the variables that were found by the type checker. The search proceeds from the initial storage by continuously taking the unbound metavariable with the highest index from the current storage and trying to refine it by using an oracle. The oracle takes as input the name of an abstract category, and it produces an expression of an atomic type that uses the same category. If the category has indices, then they are instantiated appropriately. The search procedure binds the target metavariable with the produced expression, but the expression itself can contain new metavari-

$$\frac{(x : [\![\sigma, A]\!]) :: \Gamma \vdash [\![(@|\Gamma|\ \mathit{nil}) :: \sigma, B]\!] \rightsquigarrow e}{\Gamma \vdash [\![\sigma, (x : A) \to B]\!] \rightsquigarrow \backslash x \to e} \quad \frac{(x : [\![\sigma, A]\!]) :: \Gamma \vdash [\![(@|\Gamma|\ \mathit{nil}) :: \sigma, B]\!] \rightsquigarrow e}{\Gamma \vdash [\![\sigma, (\{x\} : A) \to B]\!] \rightsquigarrow \backslash \{x\} \to e}$$

$$\frac{(x : [\![\sigma, A]\!]) :: \Gamma \vdash [\![\sigma, B]\!] \rightsquigarrow e}{\Gamma \vdash [\![\sigma, A \to B]\!] \rightsquigarrow \backslash x \to e}$$

$$\frac{\Gamma \vdash C \rhd e : [\![\sigma', C\ e'_1 \dots e'_n]\!] \quad i \leftarrow \mathit{newGuard}\ e \quad \forall j.\ |\Gamma|, i \vdash \mathit{eval}\ \sigma\ e_j \simeq \mathit{eval}\ \sigma'\ e'_j}{\Gamma \vdash [\![\sigma, C\ e_1 \dots\ e_n]\!] \rightsquigarrow ?i}$$

Figure 3.6: Proof Search Rules

ables for subgoals which are instantiated later by the procedure. The oracle can also influence the order in which the different goals are attacked by creating the metavariables in different order. We denote the action of the oracle with the statement:

$$\Gamma \vdash C \rhd e : [\![\sigma, C\ e_1 \dots e_n]\!]$$

where Γ is the current scope and C is the target category. The output from the oracle is the expression e with its type $[\![\sigma, C\ e_1 \dots e_n]\!]$. The oracle is of course nondeterministic, since for every type there are potentially infinitely many expressions, so the actual implementation uses backtracking as a way to enumerate all possibilities. Usually, we just enumerate in specific order all functions that have C as a category in the return type, and every function is applied to a sufficient number of metavariables in order to get an expression of an atomic type, i.e. a category. Later we will look closer into the definition of two oracles which implement exhaustive and random search. The oracle also gets as argument the current scope Γ, and it can choose to use a local variable instead of a global function. In this way, we also take into account the local hypotheses.

The proof search procedure itself is denoted with the statement:

$$\Gamma \vdash [\![\sigma, A]\!] \rightsquigarrow e$$

and its definition consists of the four rules shown on Figure 3.6. The first two rules handle the case when the target type is a dependent function with an explicit or implicit argument. The only difference between the two rules is that in the second case we have to produce an implicit lambda abstraction while in the first case it is

explicit. In both cases, we simply extend the scope Γ with the local variable x and its type $[\![\sigma, A]\!]$, and after that in the new context we search for proof of B. Since x appears free in B we must also extend the environment σ with $@|\Gamma|$ *nil*. The nondependent function types are covered with the third rule, and it is almost the same as the first two rules. The only difference is that the environment for B does not have to be extended.

Finally, in the third rule we have an atomic type, so we can resolve it by calling the oracle. Since the oracle is guided only by the target category and not by the values of the indices $e_1 \ldots e_n$ of the type, it can actually suggest refinements which would not lead to a type correct proof. For that reason, in the rule, we evaluate and unify the corresponding indices e_j and e'_j. On the other hand, the unification problem might be too hard, so we wrap the expression e with the guarded constant i which will be locked, if some unification has to be suspended. When this happens, then the proof search will continue with the next goal, i.e. metavariable, in the queue. The resolution of the other goals might eventually substitute the metavariable that blocked the unification, and in this case the *setMeta* predicate will resume the blocked computations. In this way, we implement dynamic rescheduling like in λProlog.

The last bit to be mentioned is the limit of the search depth. This is achieved by attaching a limit to every metavariable, where for the metavariable of the initial goal the limit is N, and it is decreased by one for every new level of subgoals. If the limit for some metavariable is zero, then the search procedure fails to refine it further and it will backtrack to find alternatives in the upper levels of the proof.

3.4.1 Random and Exhaustive Generation

The standard GF distribution provides two search strategies (oracles) – random and exhaustive generation. In principle, the user can define his/her own tailored strategies by using the low-level programming interfaces of the interpreter but since this is still experimental we will only elaborate on the predefined strategies.

The first thing that the oracle does is to select which function should be used in the refinement. In the Portable Grammar Format, for every category we maintain a list of functions which generate the same category. When we use exhaustive generation, then the oracle will suggest each function in the order in which it is listed. Since the tradition in Prolog is that the interpreter should try the clauses in the order in which they are written in the source code, we choose to adopt the same strategy, i.e. the GF compiler builds the list in source code order. Note that since the clauses in Prolog can call predicates which cause side effects (e.g.

printing or writing to the file system), the execution order is actually important for the semantics of the program. Even if the program is free from side effects, the incorrect ordering can still cause infinite loops. Fortunately none of this problems is an issue in GF, because there is no way to produce side effects in the grammar, and the infinite loops are avoided by restricting the maximal search depth. Still by controlling the order in which the functions are selected, the GF programmer can influence the overall efficiency of the proof search.

The random generation selects a function from the same list but this time the selection is randomized. Each function in the abstract syntax has some probability attached, and the randomized oracle selects the function with respect to the associated probability distribution. Unfortunately, this is still not enough to get random samples of the abstract trees if more than one sample is needed. The problem is that in the exhaustive generation if we want more than one sample, then we can simply continue the backtracking until we find a sufficient number of samples. If we do this in the random generation, then the subsequent sample will not be different enough. For instance, let say that we want to generate natural numbers in Peano representation. Suppose that the first choice of the oracle was to use the function *succ*, then we will get the partial expression:

$$\textit{succ}\ ?1$$

Now we must choose some random refinement for $?1$. Maybe this time the first choice will be *zero* and we will get the result $(\textit{succ}\ \textit{zero})$ as the first output from the proof search. If now we do backtracking, then as a second result we will get $(\textit{succ}\ (\textit{succ}\ ?2))$, but this is not different enough from the first result because they both start with *succ*. The remedy for this effect is to restart the search after every successfully found expression.

The second step in the oracle is to create metavariables for the arguments of the function, i.e. for the subgoals. Again we follow the tradition in Prolog that the subgoals should be executed in left-to-right order, but this time it must be modified in order to reflect the fact that we operate in type theory rather than in predicate calculus. If we have a function with a simple nondependent type, i.e.:

$$f : A_1 \rightarrow A_2 \rightarrow \ldots A_n \rightarrow B$$

then both the random and the exhaustive oracle will produce an expression like:

$$f\ ?(i+n)\ ?(i+n-1)\ \ldots ?i$$

i.e. here the metavariables are created in right-to-left order. Now since the proof search will continue with the unbound metavariable with the greatest index, the first subgoal to be resolved will be the first on the left.

Now suppose that f has a type with dependencies, for instance:

$$f : (x : A) \to B\ x \to Cx \to D$$

where $A, B, C,$ and D are some categories. Now it would be rather inefficient to do the proof search in the same way as for the nondependent types, since this will imply that the first subgoal to be resolved would be A. If we look at the analogous Prolog program[4]:

$$D :\!-B(x), C(x).$$

then it will be clear that the Prolog interpreter would start the search from the goal $B\ x$. In a sense, $(x : A)$ just defines the type of a local variable which in standard Prolog is omitted because the language is dynamically typed, and in λProlog it is unnecessary because its type system permits full type inference. Starting the search from the goal A is equivalent to the generation of all possible values for x followed by checking whether $B\ x$ is satisfied for the current value. Usually this is less efficient because the space for x is much wider than the space of values which satisfy the predicate B.

The strategy that we adopt is that the list of arguments for f is traversed twice, once from left to right and once from right to left. In the left-to-right direction we create the metavariables for the arguments which are bound on the type level (like x in our example) and on the way back, i.e. right-to-left, we create metavariables for the rest of the arguments. In this way we get the expression:

$$f\ ?i\ ?(i+2)\ ?(i+1)$$

Now it is clear that the first goal to be solved will be B, followed by C and finally A. Usually the solution for B or C will resolve x, so the goal A will be automatically solved. There are two exceptions when this does not happen. The first one is when there is a function which returns B but it is polymorphic in respect to the parameter, i.e. let say that we have the function:

$$g : (x : A) \to B\ x$$

[4] here we use a notation that is the opposite of the standard notation in Prolog, i.e. identifiers starting with capital letters denote predicates, while the one with small letters denote free variables.

In this case, the proof search will simply find that $(g\ ?i)$ is a suitable solution for the goal $B\ x$, i.e. we just put as argument the same metavariable that is used for the subgoal A. This does not cause any problem, and the search will simply continue with the goal $C\ x$. If this does not lead to substitution for x either, then the next goal to be solved is the metavariable $?i$ itself, so we will finally substitute x with its possible values.

The second exception is when the solution for the subsequent goals cannot be found due to some hard unification problem. In this case, the proof search will soon fall back to solving directly the metavariable $?i$ followed by resumption of the suspended unification problems. This is equivalent to running the Prolog program:

$$D :- \mathit{isA}(x), B(x), C(x).$$

where the special predicate isA will generate all possible values for x which after that will be checked for B and C. This trick is actually not uncommon in Prolog where this is used for brute force search when the target predicate does not work properly when it is applied to unbound variables. The advantage of our search strategy is that it will automatically select the best way without the need for help from the programmer.

3.5 Parsing and Dependent Types

The parser as it is described in Chapter 2 does not respect the dependent types in the abstract syntax, so it may return abstract trees which are not type correct. There are two possible solutions, either we integrate some form of type checking directly in the parser, or we type check the output from the parser, and we filter out the trees which are not type correct.

The first option is clearly more interesting, especially if it can be integrated with the incremental parser, since this can filter out some misleading predictions, when the parser is used in an authoring system for controlled languages. Unfortunately this direction hides many problems. First of all the algorithm for PMCFG parsing and the type checking algorithm are quite complex by themself, and if we combine them, the combination will be even more complex. Second the combination will suffer from serious efficiency issues, since in the intermediate stages we will have to type check parts of the chart that may be discarded later as incompatible with the rest of the sentence. Finally, it is not immediately obvious but if we want incremental type checking, then the incremental parser will be reduced to

a sophisticated version of Definite Clause Grammar (DCG) [Pereira and Warren, 1986]. This is easy to demonstrate with an example. Suppose that we have the function f:

$$\textbf{fun}\ f : (x : A) \to B\ x \to C$$
$$\textbf{lin}\ f\ x\ y = x +\!\!+\ y$$

and that the parsing with category A is ambiguous. Now in the intermediate state when the phrase for A is already consumed, we will have two or more possible values for x which we need to propagate to the type level as an argument for B. Since the type checker works with single values and not with sets, at this point we have to split the parse item for f into many items where every item will correspond to a different value for x. This is esentially equivalent to backtracking as it is used in DCG, and it brings all the problems that made DCG unattractive. Mainly, the parsing complexity becomes exponential or even infinite, when the grammar has left-recursion. Although both problems can be resolved by using tabling [Swift, 1999], this only leads to extra complications in the algorithm. Finally, even if we solve all the problems that we mentioned, the parser will still fail to filter out misleading suggestions, if this involves solving hard unification problems.

Despite the difficulties that we envisioned, it might be still possible to build useful parsing algorithm with integrated type checking, but we decided that the effort in doing this is not worth the outcome. We followed the simpler path where the parser ignores the dependent types, but the algorithm for tree extraction filters out the incorrect trees. Still as a compensation, if the user has entered a sentence which is grammatical but not type correct, then we are able to generate an appropriate error message, and we can show the error location. Similar pattern can be seen in most programming languages, i.e. the compiler parses the program in the first phase, and all other semantic analyses are done later. In contrast, if the type checker was integrated in the parser, then in the same situation the user will only get a message saying that the parsing had failed at a given location.

The rules for tree extraction (Figures 3.7 and 3.8) are very similar to the type checking rules, but this time we have to take into account that the input is not an abstract expression but a parse chart produced from the parser. At the same time the tree extraction is nondeterministic and in this it is more like a proof search where the search is constrained by the availability of items in the parse chart. The rules are transformations which simultaneously extract the trees from the chart and at the same time check their validity with respect to the constraints in the

$$\frac{\begin{array}{l} C \to f[\tau_1^{\mathcal{C}} \ldots \tau_n^{\mathcal{C}}] \\ \Gamma, \tau_1^{\mathcal{C}} \vdash \psi_F(f) : [\![\textit{nil}, \tau^{\mathcal{A}}]\!] \leadsto e_1 : [\![\sigma_1, \tau_1^{\mathcal{A}}]\!] \\ \vdots \\ \Gamma, \tau_n^{\mathcal{C}} \vdash e_{n-1} : [\![\sigma_{n-1}, \tau_{n-1}^{\mathcal{A}}]\!] \leadsto e_n : [\![\sigma_n, \psi_N(C)\ u_1 \ldots u_m]\!] \\ j \leftarrow \textit{newGuard}\ e_n \qquad \forall i. \quad |\Gamma|, j \vdash \textit{eval}\ \sigma_n\ u_i \simeq \textit{eval}\ \sigma'\ u'_i \end{array}}{\Gamma, C, [\![\sigma', \psi_N(C)\ u'_1 \ldots u'_m]\!] \vdash ?j}$$

$$\frac{i \leftarrow \textit{newMeta}\ \Gamma\ [\![\sigma, \psi_N(C)\ u_1 \ldots u_m]\!]}{\Gamma, C, [\![\sigma, \psi_N(C)\ u_1 \ldots u_m]\!] \vdash ?i} \quad C \in N^{\mathcal{C}}$$

$$\frac{(y : [\![\sigma, \psi_N(C_1)\ u_1 \ldots u_m]\!]) :: \Gamma, \tau^{\mathcal{C}}, [\![(@|\Gamma|\ \textit{nil}) :: \sigma, \tau^{\mathcal{A}}]\!] \vdash\ e}{\Gamma, C_1/C_2 \to \tau^{\mathcal{C}}, [\![\sigma, (x : \psi_N(C_1)\ u_1 \ldots u_m) \to \tau^{\mathcal{A}}]\!] \vdash\ (\backslash y \to e)} \quad y = \textit{var}(C_2)$$

$$\frac{(y : [\![\sigma, \psi_N(C_1)\ u_1 \ldots u_m]\!]) :: \Gamma, \tau^{\mathcal{C}}, [\![(@|\Gamma|\ \textit{nil}) :: \sigma, \tau^{\mathcal{A}}]\!] \vdash\ e}{\Gamma, C_1/C_2 \to \tau^{\mathcal{C}}, [\![\sigma, (\{x\} : \psi_N(C_1)\ u_1 \ldots u_m) \to \tau^{\mathcal{A}}]\!] \vdash\ (\backslash\{y\} \to e)} \quad y = \textit{var}(C_2)$$

$$\frac{(y : [\![\sigma, \psi_N(C_1)\ u_1 \ldots u_m]\!]) :: \Gamma, \tau^{\mathcal{C}}, [\![\sigma, \tau^{\mathcal{A}}]\!] \vdash\ e}{\Gamma, C_1/C_2 \to \tau^{\mathcal{C}}, [\![\sigma, \psi_N(C_1)\ u_1 \ldots u_m \to \tau^{\mathcal{A}}]\!] \vdash\ (\backslash y \to e)} \quad y = \textit{var}(C_2)$$

Figure 3.7: Extraction and Type Checking of Abstract Trees

types. We use the statement:

$$C \to f[\tau_1^{\mathcal{C}} \dots \tau_n^{\mathcal{C}}]$$

as a preconditition in the premises of the rules which ensures that the cited production exists in the chart. Here C is some concrete category, f is a concrete function, and $\tau_1^{\mathcal{C}} \dots \tau_n^{\mathcal{C}}$ are some concrete types as defined in Section 2.7. We also keep the notation from Chapter 2 which denotes the relation between concrete and abstract syntax with the mappings ψ_F and ψ_N. Like in the type checker, we use statements for term unification and for creation of metavariables, but we also need to introduce two new kinds of statements. The main statement:

$$\Gamma, \tau^{\mathcal{C}}, [\![\sigma, \tau^{\mathcal{A}}]\!] \vdash\ e$$

declares that the expression e was extracted from the chart by starting from the concrete type $\tau^{\mathcal{C}}$. The abstract type of the expression is $[\![\sigma, \tau^{\mathcal{A}}]\!]$, and it is defined in the scope Γ. For example, in order to extract all the trees for the start category S, we must look up all passive items of the form $[^n_0 A; 0; A']$ where $\psi_N(A) = S$ and then derive the statement:

$$\mathit{nil}, A', S \vdash\ e$$

If this is successful, then e is an abstract syntax tree for the parsed sentence.

Furthermore, the rules that define the extraction rely on the additional statement:

$$\Gamma, \tau^{\mathcal{C}} \vdash\ e_1 : [\![\sigma_1, \tau_1^{\mathcal{A}}]\!] \rightsquigarrow e_2 : [\![\sigma_2, \tau_2^{\mathcal{A}}]\!]$$

which takes the expression e_1 with its type $[\![\sigma_1, \tau_1^{\mathcal{A}}]\!]$ and produces another expression e_2 of type $[\![\sigma_2, \tau_2^{\mathcal{A}}]\!]$, where e_1 is applied to the result of the tree extraction from the concrete type $\tau^{\mathcal{C}}$. When e_2 is being constructed, the rules for the helper statement take care for the update of the environment, i.e. the construction of σ_2, and they also use implicit or explicit application as needed in e_2.

The extraction rules are listed on Figure 3.7. The first rule is the most complex and the most central one, since it describes the conversion of a whole production to an abstract expression. The first precondition in the rule states that the chart must contain the production $C \to f[\tau_1^{\mathcal{C}} \dots \tau_n^{\mathcal{C}}]$. If this is the case, then we transform the concrete function f into the abstract function $\psi_N(f)$, whose type $\tau^{\mathcal{A}}$ we can retrieve from the grammar. After that we apply $\psi_N(f)$ to the necessary number of arguments until we get a fully saturated function application. We use the helper

statement to express the application, i.e. we start from $\psi_F(f) : [\![nil, \tau^{\mathcal{A}}]\!]$ and after n applications we get $e_n : [\![\sigma_n, \psi_N(C)\ u_1 \ldots u_m]\!]$. Since the GF compiler ensures that the result category C in the production is mapped to the result abstract category in the type signature for f, we can be sure that the category for e_n will be exactly $\psi_N(C)$. However, the compiler ignores the indices of the category, so now we must use unification in order to ensure the agreement, i.e. we must unify u_i with u'_i for every i. Again since the unification might be too hard, we guard the expression e_n with the metavariable $?j$, and we return $?j$ as a final result instead of e_n.

The first rule is applied only if C is a newly generated category. If it is not, i.e. if $C \in N^{\mathcal{C}}$, then we have an erasing rule, i.e. the linearization of some argument of some function is suppressed. As we said in Section 2.3.7, in this case we must stop the extraction and simply generate fresh metavariable. This is implemented by the second rule on Figure 3.7.

The last three rules on the figure deal with the higher-order syntax. The only difference between the rules is the choice of dependent vs nondependent type and implicit vs explicit function. In all cases, we extend the scope Γ with a new variable and we proceed with the extraction, but also we wrap the final expression with implicit or explicit lambda abstraction. The variable for the abstraction is taken from the category C_2, i.e. there must be a production:

$$C_2 \to h[N], \qquad h \in \mathit{lindef}(C_2)$$

where N is a category produced by the LITERAL rule in the parser, which encodes the variable name. When the type of the function is dependent, then in the recursive step we also extend the environment σ.

Finally, the helper statement in the definition of the first extraction rule is defined with three simple rules on Figure 3.8. Basically, the rules take the partial expression e_1 and retrieve the type of the next argument from the type of e_1. Based on the new type, we extract the argument itself, i.e. e_2, and we apply e_1 to e_2 with either implicit or explicit application.

At this point it might be still unclear how it is possible to detect the location of a type error in the source text. The key is that the only case when a type error might occur is in the first rule on Figure 3.7 which fails if the unification of the indices u_i and u'_i is not possible. In this case we find the source location from the category C. Remember from Section 2.4 that each phrase in the parsed sentence is uniquely identified by its concrete category. We use this to report the error location if the sentence is parseable but the semantic restrictions are not satisfied.

$$\frac{\Gamma, \tau^{\mathcal{C}}, [\![\sigma, \tau_1^{\mathcal{A}}]\!] \vdash e_2}{\Gamma, \tau^{\mathcal{C}} \vdash e_1 : [\![\sigma, (x : \tau_1^{\mathcal{A}}) \rightarrow \tau_2^{\mathcal{A}}]\!] \rightsquigarrow e_1\ e_2 : [\![\textit{eval}\ \sigma_\Gamma\ e_2 :: \sigma, \tau_2^{\mathcal{A}}]\!]}$$

$$\frac{\Gamma, \tau^{\mathcal{C}}, [\![\sigma, \tau_1^{\mathcal{A}}]\!] \vdash e_2}{\Gamma, \tau^{\mathcal{C}} \vdash e_1 : [\![\sigma, (\{x\} : \tau_1^{\mathcal{A}}) \rightarrow \tau_2^{\mathcal{A}}]\!] \rightsquigarrow e_1\ \{e_2\} : [\![\textit{eval}\ \sigma_\Gamma\ e_2 :: \sigma, \tau_2^{\mathcal{A}}]\!]}$$

$$\frac{\Gamma, \tau^{\mathcal{C}}, [\![\sigma, \tau_1^{\mathcal{A}}]\!] \vdash e_2}{\Gamma, \tau^{\mathcal{C}} \vdash e_1 : [\![\sigma, \tau_1^{\mathcal{A}} \rightarrow \tau_2^{\mathcal{A}}]\!] \rightsquigarrow e_1\ e_2 : [\![\sigma, \tau_2^{\mathcal{A}}]\!]}$$

Figure 3.8: Helper for Extraction of Abstract Trees

Chapter 4

Frontiers

In this chapter, we discuss ideas that are still on the research frontline and as such they are still unfinished. In some cases, we even speculate a bit and propose solutions that are still not experimentally proven, but at least we justify our opinion with experimental results reported by other researchers for similar problems. Our main goal is not to establish firm procedures but only to show two interesting research directions which we see as an important continuation of the work that we reported in the previous chapters.

The first and perhaps the most important direction is to scale up from a framework for controlled languages to a more general framework that is equally well suited for applications that require robust processing of unrestricted text. The advantage of the controlled languages is that their syntax and semantics are very well defined, which makes it possible to do reliable processing. The price, however, is that the language is restricted which makes it hard to process already existing content, unless if it is already tailored for the particular application. On the other hand by using statistical methods, it is possible to build robust applications that can process unrestricted text. Unfortunately this is reasonable only under the assumption that certain percentage of errors is permitted.

The standard workflow in GF is to define application specific grammars which capture the precise semantics of the different domains. Furthermore, since the development of new grammars from scratch is a costly process, the GF community has developed a library of resource grammars [Ranta, 2009] that can be reused in the application grammars by automatic specialization. In other words, the application grammar specifies the semantics and the pragmatics of the domain while the resource grammar provides reusable syntax which matches the syntax of the host natural language. This standard workflow guarantees results with high quality and

low development cost but it is viable only for narrow domains.

An alternative workflow is to start directly from the resource grammar and to add on top of it some statistical model which can be used for disambigution in the otherwise highly ambiguous grammar. Since the resource grammars have wide coverage, this will give us robust processing but at the cost that the statistical model might fail to select the right interpretation. This new direction was opened only recently, thanks to the development of the optimization techniques and the new parsing algorithms that are the subject of this monograph.

In Section 4.1, we will present our preliminary work on parsing the Penn Treebank [Marcus et al., 1993] with the English resource grammar, and we will suggest how this can lead to the development of wide coverage robust parsers.

The second interesting research direction is how to better utilize the reasoning capabilities of the framework. Most GF applications use only simply typed abstract syntax, and if more complicated pre- or postprocessing is needed then it is done outside of the grammar. In Section 4.2, we take advantage of the improved support for dependent types, and we show how it is possible to perform two directional semantic processing, i.e. we can either do semantic analysis of an already parsed sentence, or we can do natural language generation from a given semantic representation. The processing is two dimensional in the sense that the user specifies the relation between syntax and semantics only once, and the analysis and the generation are given for free by using different services in the interpreter.

Although semantic processing on a large scale is in principle possible [Bos et al., 2004], we consider this as a low-priority task since the wide coverage parser have to be finished first. Still the semantic capabilities can already be utilized in non-trivial application grammars.

4.1 Statistical Parsing

The resource grammars library is a collection of grammars whose initial design purpose was to serve as basis for building application grammars rather than to be used for direct parsing. Despite this we showed in Section 2.3.9 that they are not only usable but actually the parser performs quite well even on this scale. A natural question is how much we can achieve by using the resource grammars for parsing free text. So far we can only give a partial answer by evaluating the English Resource Grammar in combination with a lexicon of about 40 000 lemmas derived from the Oxford Advanced Learners Dictionary. The goal of the evaluation is twofold. From one side we measure the coverage of the grammar, and from

another we prepare the training data that will be used for building a statistical disambiguation model compatible with the resource grammar. Similar experiments were successful for other formalisms, i.e. Riezler et al. [2002] for LFG, Miyao and Tsujii [2005] for HPSG, Clark and Curran [2007] for CCG, so we can expect that this would work well for GF too.

The final parser will produce partial results in a way similar to Riezler et al. [2002] for sentences that are not completely covered by the grammar. At this point we do not consider extensions to the resource grammar that would cover the missing constructions from the Penn Treebank because this would still not guarantee that the grammar will work on unseen text. Instead we expect the necessary robustness to come from the parsing algorithm and we will give some hints about how this could be done. Possible extensions of the grammar can always be added later in a separate project. The fully automatic extraction of GF grammar from a treebank is also not an option because we want to preserve the abstract syntax of the resource grammar. This essentially rules out the route taken by Miyao and Tsujii [2005] and Clark and Curran [2007].

The evaluation is done by converting all sentences from sections 2–21 in the Penn Treebank [Marcus et al., 1993] to partial abstract syntax trees. The trees are partial in the sense that whenever we encounter a construction in the treebank that is compatible with the grammar, we generate the corresponding snippet of abstract tree, and if this is not possible, then we generate a metavariable which replaces the construction. In this setting, we measure the coverage as the percentage of nodes in the abstract trees that are filled in with function names instead of metavariables.

The first problem in the conversion is that the central concept in GF is the abstract syntax tree instead of the parse tree. There is some discrepancy between the two concepts and the conversion is possible only if the parse tree follows the same linguistic theory as the one used in the grammar. Unfortunately, this is not the case for the English resource grammar and the Penn Treebank. For instance the resource grammar has more fine grained part of speech tags and deeper syntactic analyses. Ideally we would recover the missing information by parsing every sentence in the treebank and then comparing the alternative analyses with the annotations. In practice, however, this is not feasible because there are far too many alternatives, and most of the sentences cannot be fully parsed with the grammar.

Our solution is to parse not the plain sentence but the different syntactic levels of the already annotated sentence. If we take as an example the sentence on Figure 4.1, then the structure at the top level is:

```
(S (NP ...) (VP ...))
```

```
(S
  (NP-SBJ (NNP BELL) (NNP INDUSTRIES) (NNP Inc.) )
  (VP (VBD increased)
    (NP (PRPS its) (NN quarterly) )
    (PP-DIR (TO to)
      (NP (CD 10) (NNS cents) ))
    (PP-DIR (IN from)
      (NP
        (NP (CD seven) (NNS cents) )
        (NP-ADV (DT a) (NN share) )))))
```

Figure 4.1: An annotated sentence from Penn Treebank

which we can immediately map into the abstract syntax tree:

$$\mathit{UseCl}\ ?1\ ?2\ (\mathit{PredVP}\ ?3\ ?4)$$

Here the metavariables $?3$ and $?4$ must be bound with the output from the conversion of the `NP` and `VP` phrases, which is done on the next level. Similarly, the metavariables $?1$ and $?2$ correspond to the abstract representation of the tense and the polarity of the sentence and are also extracted from the `VP` phrase. Since the conversion is guided by the annotations and not by the plain text, we completely avoid the generation of irrelevant analyses.

The example also demonstrates the robustness of the conversion. If we are not able to find a matching abstract syntax tree for some level, then we simply convert all children and combine the outputs with a metavariable. For instance the grammar does not have a rule for combining two NP phrases into another NP phrase but still the conversion is able to produce:

$$\begin{array}{l}(?34\ (\mathit{DetCN}\ (\mathit{DetQuant}\ \mathit{IndefArt}\ (\mathit{NumCard}\ \ldots))\ (\mathit{UseN}\ \mathit{cent_N}))\\ \quad (\mathit{DetCN}\ (\mathit{DetQuant}\ \mathit{IndefArt}\ \mathit{NumSg})\ (\mathit{UseN}\ \mathit{share_N})))\end{array}$$

for the phrase "seven cents a share", i.e. it converts each of the embedded phrases and combines them into one phrase by applying a metavariable. The new phrase is further on embedded in the abstract syntax for the prepositional phrase "from seven cents a share". The full abstract syntax tree for the example sentence is shown in Figure 4.2.

The exact transformation rules are implemented in Haskell and encoded by

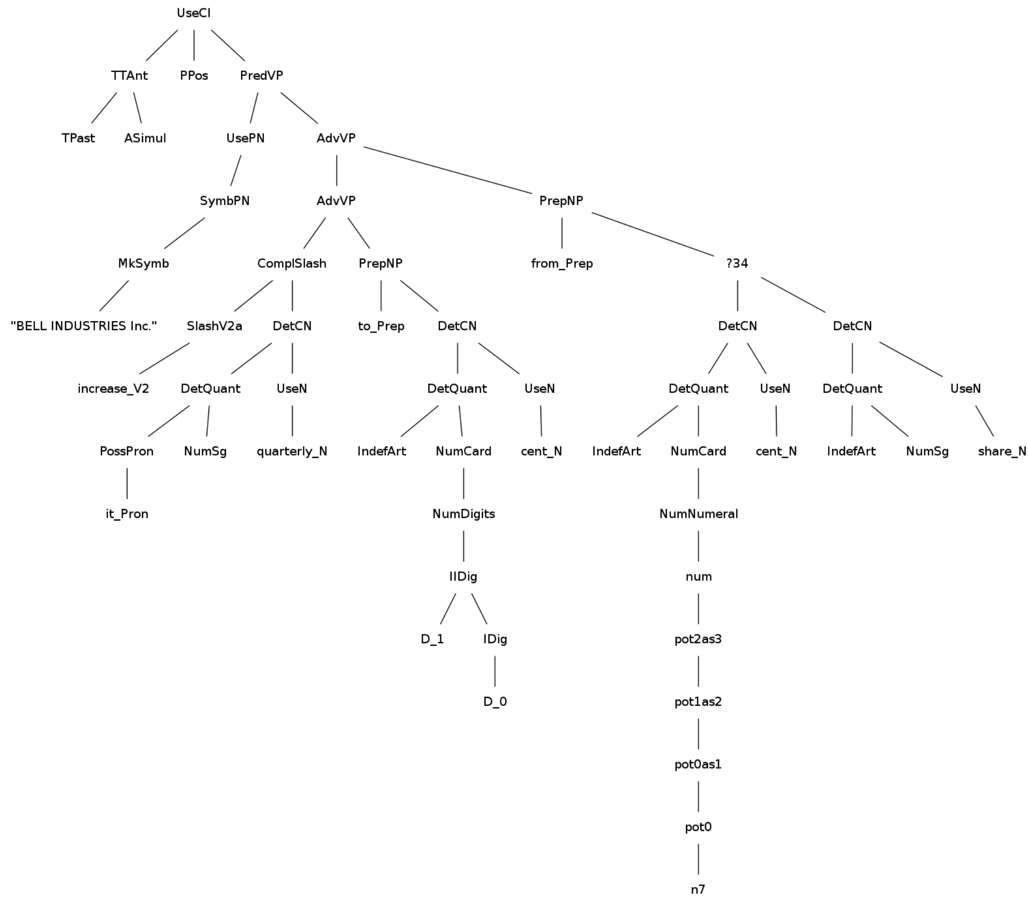


Figure 4.2: The abstract syntax tree for the example on Figure 4.1

using parser combinators [Hutton, 1992][1]. The only complication is that while the traditional parser combinators are used for parsing sequences of tokens, in our case we have tree structures. Still in most cases we are only interested in the children of one particular node and since they are ordered we can continue to use traditional parsing techniques. For instance the transformation for a sentence composed of one `NP` and one `VP` phrase can be encoded as the following Haskell code:

```
"S" :-> do np <- cat "NP"
           vp <- cat "VP"
           return (PredVP np vp)
```

Here the operator `:->` assigns some rule to the category `S` and every time when we see the same category in the treebank, we will fire the rule associated with it. The rule processes the children of the node and combines the results by using the `return` combinator. In this case, the processing consists of calling the combinator `cat` twice, once for `"NP"` and once for `"VP"`. Each time, the `cat` combinator checks that the current child has the corresponding category and applies recursively the transformation rule for its own children. If the application of the rule is successful, then `cat` returns the output from the rule. If it is not, then the result is a metavariable applied to the transformation of the grandchildren.

This first example is oversimplified and it does not work even in this simple case because it does not let us to extract the tense and the polarity of the sentence. The solution is to refactor the rule into:

```
"S" :-> do np <- cat "NP"
           (t,p,vp) <- inside "VP" pVP
           return (UseCl t p (PredVP np vp))
```

Now we have a new combinator called `inside` which also checks that the current child is `"VP"` but instead of calling the generic rule for transformation of verb phrases, it calls the custom rule `pVP` which returns both the transformed phrase and its tense and polarity. While if we were using only `cat`, our transformation rules will be equivalent to context-free grammar, with the introduction of `inside` we get both some context-sensitivity and the ability to go deeper in the nested levels of the parse tree.

---

[1] In functional programming, the parser combinators play the same role as the use of DCG in Prolog.

So far we have used `inside` as a way to get more information for the phrase, but we can also use it when the annotation structure of the treebank is incompatible with the abstract syntax of the resource grammar. For instance the grammar does not have `SBAR` category like in the treebank, and we must do the modelling differently. For example the verb `said` takes as object another sentence but in the treebank the object is annotated as a sentence wrapped in an `SBAR` phrase:

```
(S (CC But)
  (NP-SBJ (NNP Mr.) (NNP Lane) )
  (VP (VBD said)
    (SBAR (IN that)
      (S
        .....
               ))))
```

Still we can do the transformation by using rule like:

```
"VP" :-> do v <- pV "VS"
            s <- inside "SBAR"
                    (do inside "IN" (word "that")
                        cat "S")
            return (ComplVS v s)
```

where we go inside the `SBAR` annotation and look for an `S` annotation which after that is processed by using `cat`. The rule also shows the usage of the combinator `word`, which is used when we want to check for the presence of specific words.

Here we called the custom rule `pV` instead of calling the generic rule for verbs, because we need to select the verb with the right valency which is controlled by the argument `VS`. Currently the transformation rules recognize the following verb types:

V	intransitive verb
V2	transitive verb
VV	verb with verb phrase as object
VA	verb with adjectival phrase as object
VS	verb with sentence as object
V2V	verb with one noun phrase and one verb phrase as object

and the only way to distinguish the different types is to look in the context. If for instance the object is a sentence, then in the abstract syntax we should use the

lexical constant:

$$\textbf{fun}\ \mathit{say_VS} : \mathit{VS};$$

instead of *say_V2* or some other valency type. Note that currently we do not recognize verb complements that are prepositional phrases. Instead we always attach the prepositional phrases as modifiers.

The selection of the lexical constants itself is done by using the combinator *lemma* which takes as argument the name of the abstract category, i.e. *VS* in our case and the name of a field in the linearization type of the category, and tries to find a lexical entry in the grammar which contains the current word in the parse tree. For instance the word `said` is processed by the call:

lemma ”VS” ”s VPast”

This will search all constants of type *VS* and will find that *say_VS* has the given word in the right place in its linearization table, i.e. its table is:

s VInf	:	say
s VPres	:	says
s VPPart	:	said
s VPresPart	:	saying
s VPast	:	said

This demonstrates yet another difference between the annotations in the treebank and the abstract syntax trees from the grammar. The lexical entries in the grammar are tuples which contain all inflection forms of the word, while in the treebank we have the particular word form and its part of speech tag. In fact, the part of speech tag in Penn Treebank corresponds to the pair of abstract category and field name. For instance, we have the relation:

$$\mathit{VBD} \leftrightarrow (\mathit{VS}, \text{"s VPast"})$$

Similar relations are the basis for the generic transformation rules for most of the lexical units in the treebank. For instance the generic rules for the adjectives are:

```
"JJ"  :-> do a <- lemma "A" "s (AAdj Posit Nom)"
             return a
"JJR" :-> do a <- lemma "A" "s (AAdj Compar Nom)"
             return a
"JJS" :-> do a <- lemma "A" "s (AAdj Superl Nom)"
             return (OrdSuperl a)
```

In addition, there are few more combinators which allow more flexible control on the transformation. For instance, the combination `r1 ‘mplus‘ r2` allows us to try the rule `r1` first, and if it fails to continue with `r2`. The combinations `many r` and `many1 r` act similarly to the Kleene star and plus operators, i.e. they repeat the application of rule `r`, and they return as result a list with the outputs produced from `r`.

With the help of the combinators we encoded the full transformation from the Penn Treebank to the English resource grammar in only 800 lines of code. Since the transformation is robust we managed to transform all sentences but of course for some of them the output tree is only partial. We counted that in average 91.75% of the nodes in the trees are filled in with functions from the grammar, and the remaining 8.25% are metavariables. As a whole there are 39832 trees of which 1633 trees were fully converted, 4668 contained only one metavariable, 4803 had two metavariables and the rest had more than two. In average there are 4.91 metavariables per tree.

We looked at some of the incomplete trees, and we identified three main reasons for incomplete matching. In many cases, simply the grammar did not have the corresponding syntactic rule. In other cases, there were inconsistencies in the treebank, and although the conversion would be possible in principle, our automatic translation simply failed. We fixed some of the inconsistencies by adding extra rules in the transformation, but this is not always possible. One trivial example is the word `many` which is sometimes annotated as adjective `JJ`, sometimes as adverb `RB`, and only occasionally as a determiner `DT`. In the resource grammar, it is always considered to be determiner, and we fixed this case by adding special rules for `many`. We also noticed other cases of inconsistencies but since they are more rare and less regular we did not try to fix all of them. Finally, we are aware that we might have missed some transformation patterns just because they are rare and we have not noticed them. It is not possible to evaluate the impact of the different factors because then we would have to check all trees manually which would take too long time.

Since we already have transformation rules from the Penn Treebank format to abstract syntax trees, in fact, we do not need anything else in order to do robust parsing with wide coverage. We can use any existing statistical parser like the Stanford parser or the Collins parser, and then we can convert the output to an abstract syntax tree. This is not completely satisfactory, however, since in this way we bypass the grammar completely, and we cannot take advantage of it.

The first missing piece for making a standalone wide coverage GF parser is the disambiguation model. The grammar does not fully cover the treebank but still

it is pretty close, and we have collected enough material for training the model. The framework already has a simple statistical model where each function in the abstract syntax has a fixed probability. Essentially this is equivalent to the model of probabilistic context-free grammar except that the grammar itself is context sensitive. Although the limitations of the model for statistical parsing are well known, we have not tried to replace it yet because we inherited it from earlier versions of GF, and, as we will show, it might work well if some other theoretical issues are solved first.

We computed the probabilities of the functions from the treebank, and we did evaluation on the longest sentence with a fully reconstructed abstract syntax tree:

> In last month's survey a number of currency analysts predicted that the dollar would be pressured by a narrowing of interest rate differentials between the U.S. and West Germany.

Although the sentence is in principle fully parsable, the tree extraction failed with out of memory error, since in the default setting the parser is trying to extract all abstract syntax trees. If instead we extract only the most probable one, then we get it immediately. Still the tree that we got is not the same as the one in the treebank, but at least it has the right part of speech tags. For comparison, we looked at the first 50 000 trees that were produced by the parser without ranking, and we found that none of them had even the right part of speech tags. This is not a big surprise since all state of the art parsers are using dependency based disambiguation models. Our context-free model seems to be too simplified.

For example, Collins [2003] recommends in his Model 1, that the context-free rules from the treebank need to be first binarized and after that the probability of every binary rule must be conditionalized on the head of the phrase. The binarization is not necessary in our case since most abstract functions in the resource grammar are already at most binary. In fact, our main concern with Penn Treebank was that the trees are too shallow, so we had to convert them to deeper structures by parsing. The binarization procedure in Model 1 serves similar purpose. The only missing piece is the head conditional probability.

Contrary to some other formalisms GF does not have built in concept of head, but this can be easily reintroduced. What we need is an additional configuration which specifies the roles of the different arguments of a given function. For instance, we could have:

$$PredVP\ subj\ head$$

which specifies that the second argument of *PredVP* is the head of the clause, and the first one is the subject. In fact, the GF engine has a service which from an abstract syntax tree generates dependency tree in the format of the MALT parser [Nivre, 2006], and this service requires configuration which is exactly the same as the one above.

The configuration can be used from the disambiguation model too. The head of a phrase can be retrieved by following all arguments marked with the label *head* until we reach a leaf. The probability of the subtree for the non-head arguments should be conditionalized, then, on the head of the phrase.

We have not implemented this dependency based model yet because first of all the project is still in progress, and second it is not clear whether we need to introduce a new mechanism at all. It might be possible to represent the dependencies in Model 1 by just using the dependent types that we already have. For instance we could have another signature for *PredVP*:

$$\textbf{fun}\ \mathit{PredVP} : (v : V) \rightarrow \mathit{NP}\ v \rightarrow \mathit{VP}\ v \rightarrow S$$

where now the noun phrase is clearly dependent on some verb, and we can ensure that the verb is the head of the verb phrase by changing the definition of *UseV*:

$$\textbf{fun}\ \mathit{UseV} : (v : V) \rightarrow \mathit{VP}\ v$$

In this way we can enforce a dependency relation between the head of the noun phrase and the head of the verb phrase.

Whether or not we want to express these dependencies in the abstract syntax, it is not clear yet. From one side this makes the abstract syntax more complicated, from another there are still unsolved theoretical issues about how the statistical model should interact with the dependent types. Currently the probability of a tree is computed as the product of the probabilities of all functions in the tree, but this ignores the fact that the dependencies might enforce a that certain function must or cannot be used in certain place. A more accurate probability estimation must take into account the types of the functions. An external modelling of the conditional probabilities is clearly simpler, but in fact the conditional probabilities reflect some implicit semantics which, as we will see in the next section, is best expressed with dependent types.

Apart from the disambiguation model, the only other missing piece is to make the parser more robust when it encounters unknown syntactic constructions. This can be done either by changing the grammar to allow fragmentary parsing like in Riezler et al. [2002] or by letting the parser to “invent” such rules on the fly when the parsing cannot continue in the normal way.

4.2 Montague Semantics

Most GF applications use simply typed abstract syntax, which is enough to define the syntax of the language but from semantic point of view is very shallow. It is true that even with the simple types, the application grammar can encode some semantic differences, i.e. it can have categories like *Food* and *Drink* instead of the purely syntactic category *NP*, but beyond that it is difficult to do deeper analyses without dependent types. A notable example for the application of the Montague [1973] semantics is the grammar in Ranta [2004a] which implements a small fraction of English syntax together with its semantics.

Basically, there are two ways in GF to link the syntax with the semantics of a sentence. Either we define an interpretation function:

$$\textbf{fun}\ iS : S \rightarrow Prop$$

which maps every tree of type S into some logical formula, i.e. an object (a tree) of type *Prop*, or we make the sentence category S dependent on the semantic representation of the syntax tree:

$$\textbf{cat}\ S\ Prop$$

Ranta [2004a] in particular used a mixed representation where some of the syntactic categories depend on semantic expressions which can contribute to the syntactic disambiguation. The rest of the semantics is computed by using interpretation functions similar to iS.

The advantage of the first alternative is that the semantics is separated from the syntax, and in principle it is possible to attach different semantics to one and the same abstract syntax. Furthermore, it is possible to parse even sentences which are semantically meaningless, since the semantic processing is optional, i.e. it is performed only if we compute the application of iS.

In the second alternative, the semantic representation is computed automatically by type checking the trees during the tree extraction. For example, if the *PredVP* function is redefined as[2]:

$$\begin{aligned}\textbf{fun}\ PredVP : & (\{np\} : (Ind \rightarrow Prop) \rightarrow Prop) \rightarrow \\ & (\{vp\} : Ind \rightarrow Prop) \rightarrow \\ & NP\ np \rightarrow VP\ vp \rightarrow Cl\ (np\ vp);\end{aligned}$$

[2]The full source code for a fragment of the English resource grammar complemented with Montague semantics is available in the folder "examples/nlg" in the GF distribution (http://www.grammaticalframework.org/examples/nlg/).

then the type checker will compute the interpretations *np* and *vp* for the noun phrase and the verb phrase, and will combine them to compose the interpretation of the whole clause. This approach is similar to HPSG [Pollard and Sag, 1994] where the semantic processing is an integral part of the whole analysis. For instance, if we parse the sentence "somebody loves everybody" with the semantic grammar, then we get an abstract syntax tree which is almost the same as the abstract syntax tree in the resource grammar but now we also get the semantic representation as an additional argument to *UseCl*[3]:

```
> p "somebody loves everybody"
UttS (exists (\v0 -> forall (\v1 -> love v0 v1)))
     (UseCl PPos (PredVP somebody_NP
                         (ComplSlash (SlashV2a love_V2)
                                     everybody_NP)))
UttS (forall (\v0 -> exists (\v1 -> love v0 v1)))
     (UseCl PPos (ComplClSlash (SlashVP somebody_NP
                                        (SlashV2a love_V2))
                               everybody_NP))
```

Note that the similarity is only on the surface. In reality, the abstract syntax in the semantic grammar is much more complex, but its complexity is hidden through the usage of implicit arguments. For instance, the *np* and *vp* arguments of *PredVP* are made implicit which hides the semantics of the embedded phrases. Without the usage of implicit arguments the abstract syntax trees quickly become too verbose and hard to use in practice. The shortcomings of the earlier GF versions are also exemplified in Ranta [2004a] where the full abstract tree for the donkey sentence:

> if a man owns a donkey he beats it

is written in seven lines, while his shortened version is only two lines long. The short version is also what we would get if implicit arguments were used.

Since the semantic processing is now integrated, semantically meaningless sentences are rejected which is both advantage and disadvantage depending on the application. The main advantage of our representation, however, is that now it is possible to use proof search to find all sentences with fixed semantics. For instance

[3]Here we get two abstract syntax trees, since we handle the quantifier scope ambiguity by making the grammar ambiguous. This is the same approach as in Ranta [2004a], which is inspired by Steedman's combinatory categorial grammar [Steedman, 1988]. An alternative solution would be to keep the grammar unambiguous and to replace the fully instantiated logical formula with some underspecified representation.

in the GF shell, we can use the `gt` command to generate all abstract trees with the same semantics, for example `(and (smart john) (smart mary))`, and after that we can linearize each tree with the command `l`:

```
> gt -cat="S (and (smart john) (smart mary))" -depth=8 | l
John is smart and Mary is smart
John and Mary are smart
```

The search correctly identifies that there are two ways to express the same meaning. Either we use sentence level conjunction, or we can use the shorter noun phrase conjunction since the verb phrase is the same.

In general, the generation from arbitrary logical formula to the abstract syntax of the resource grammar is an expensive operation, but it can be rather cheap in more restricted domains. For instance, Dannélls [2010] defines a grammar for description of museum objects where the final goal is to generate the descriptions from OWL [W3C, 2004] data set. Although Dannélls defined the transformation from OWL to the abstract syntax of the grammar as an external process, we found that the transformation is trivial to define in GF, if the grammar uses the OWL description as the semantic representation of the natural language.

The interaction between GF grammars and OWL ontologies will be studied further in the ongoing project MOLTO (www.molto-project.eu), but we already see the dependent types as a general and powerful tool which was not well exploited yet due to limitations in the implementation. We believe that the improvements in the type checking algorithm and the introduction of implicit arguments make the dependent types easier to use. Further improvements, however, might be needed when we gain more experience in different applications. In particular, more experiments in natural language generation and development of controlled languages with embedded semantics will give useful feedback for the further development of the framework. Last but not least the development of wide coverage GF parsers could at the end allow large scale semantic processing in GF if we complete the semantics for the whole resource grammar [Bringert, 2008].

Chapter 5

Conclusion

This final chapter is a summary of the contributions of this monograph in the context of GF and Natural Language Processing in general.

First of all the GF framework always worked well on small scale but going out of the small scale was tremendously difficult due to the high computational demands that large scale grammars put on. The development of low-level grammar representation that can be efficiently executed is a major step towards scaling the framework to open-domain text. In our experience, for many applications, the execution is orders of magnitude faster than what it was before. This level of improvement did not happen evolutionary, in fact the whole runtime engine was replaced with a new one that is fine tuned and tested with large grammars.

Furthermore, the runtime engine is now clearly separated from the compiler which simplifies the development of standalone applications. The grammars are represented in a compact and portable format which makes it possible to reuse the grammars on different platforms and from different programming languages.

The improvements in the engine and the accumulated development in the resource grammars library led GF to a turning point where it is feasible to analyse almost unrestricted text. Still the liberation of GF from constrained language will require more work on robustness and disambiguation. The engine already has a simple probabilistic model, and developers can achieve some robustness by tweaking the grammar and by playing with the parsing API but still there is a lot to be done. In this context, the present work is more like an important milestone rather than a complete solution.

The improvement of the efficiency was always our primary motivation but as it happened another side effect of the new algorithms was much better utilized for the time being. While working with controlled languages it is always an issue for the user how to structure the input in a way understandable to the computer. The incremental nature of the new GF parser allowed the development of new user interfaces [Bringert et al., 2009][An-

gelov and Ranta, 2010] where the system helps the user to author grammatically correct content by showing suggestions. The new interfaces are usually more intuitive than the old syntax editors [Khegai and Ranta, 2003], where the content is created hierarchically phrase by phrase, but in some cases it is an advantage to see both perspectives. It is possible to have both but this requires some more engineering on the user side.

The automatic reasoning in GF is not really a new concept since the type system had dependent types from the very beginning. Unfortunately the full power of the dependent types has never been exploited to the limit. The reason for this was partly because of the poor performance of the proof search and partly because without the support for implicit arguments the development of any non-trivial logic in the abstract syntax becomes too cumbersome. In the development of the GF engine we started from scratch and we took advantage of the experience from the development of Agda. The efficiency of the proof search was a secondary goal in this case but still we followed the design of λProlog which is well developed and gives good performance. Unfortunately our implementation still lags far behind, mostly because we use direct interpretation while the λProlog programs are first compiled to low-level byte code which is after that executed by a well optimized interpreter in C. An integration of GF with λProlog will make the dependent types first class citizens in the framework.

In the wider context of natural language processing, this monograph contributes with a number of new algorithms and ideas that can be reused in other frameworks. The application of models that go beyond context-free grammars is popular in both statistical and rule based engines. In that sense, the monograph is interesting as a non-trivial application of Parallel Multiple Context-Free Grammars.

Appendix A

Portable Grammar Format

So far, we talked about the Portable Grammar Format (PGF) as an abstract concept, but in the actual implementation, we store the compiled grammars in a concrete file format. This appendix is the reference for the exact format, and it includes details which were intentionally skipped in the previous chapters because they were irrelevant to the main algorithms.

The format described here is a version 1.0 and is produced by GF 3.2. In case, if the format have to be changed in the future, the implementations should maintain backward compatibility.

The GF compiler will dump any PGF file into textual representation with a syntax close to what we used in the different chapters, if it is requested to produce the format `pgf_pretty`:

```
> gf -make -output-format=pgf_pretty MyGrammar.pgf
```

Basic Types

The Portable Grammar Format is a binary format where the structures of the grammar are serialized as a sequence of bytes. Every structure is a list of sequentially serialized fields, where every field is either another structure or has a basic type. The allowed basic types are:

- Int8 - 8 bits integer, with sign, represented as a single byte.
- Int16 - 16 bits integer, with sign, represented as a sequence of two bytes where the most significant byte is stored first.
- Int - a 32 bits integer with sign encoded as a sequence of bytes with variable length. The last bit of every byte is an indication for whether there are more bytes left. If

the bit is 1, then there is at least one more byte to be read, otherwise this is the last byte in the sequence. The other 7 bits are parts of the stored integer. We store the bits from the least significant to the most significant.

- String - a string in UTF-8 encoding. We first store as Int (a variable length integer) the length of the string in number of Unicode characters and after that we add the UTF-8 encoding of the string itself.

- Float - A double precision floating point number serialized in a big-endian format following the IEEE754 standard.

- List - Many of the object fields are lists of other objects. We say that the field is of type $[Object]$ if it contains a list of objects of type $Object$. The list is serialized as a variable length integer indicating the length of the list in number of objects, followed by the serialization of the elements of the list.

PGF

The whole PGF file contains only one structure which corresponds to the abstract structure $\mathcal{G}$ from Definition 1 in Section 2.1. The structure has the following fields:

type	description
Int16	major PGF version, should be 1.
Int16	minor PGF version, should be 0.
[Flag]	global flags
Abstract	abstract syntax
[Concrete]	list of concrete syntaxes

If PGF is changed in the future, the version in the first two fields should be updated. The implementations can use the version number to maintain backward compatibility.

Flag

The flags are pairs of a name and a literal and store different configuration parameters. They are generated by the compiler and are accessible only internally from the interpreter. By using flags we can add new settings without changing the format.

type	description
String	flag name
Literal	flag value

Abstract

This is the object that represents the abstract syntax $\mathcal{A}$ (Definition 2, Section 2.1) of a grammar. The name of the abstract syntax is the name of the top-level abstract module in the grammar. The start category is specified with the flag *startcat*.

type	description
String	the name of the abstract syntax
[Flag]	a list of flags
[AbsFun]	a list of abstract functions
[AbsCat]	a list of abstract categories

Note: all lists are sorted by name which makes it easy to do binary search.

AbsFun

Every abstract function is represented with one AbsFun object.

type	description
String	the name of the function
Type	function's type signature
Int	function's arity
Int8	a constructor tag: 0 - constructor; 1 - function
[Equation]	definitional equations for this function if it is not a constructor
Float	the probability of the function

The constructor tag distinguishes between constructors and computable functions, i.e. we can distinguish between this two judgements:

- constructor: **data** $f : T$
- function: **fun** $f : T$

If this is a function, then we also include a list of definitional equations. The list can be empty which means that the function is an axiom. In the cases, when we have at least one equation then the arity is the number of arguments that have to be known in order to do pattern matching. For constructors or axioms the arity is zero.

AbsCat

Every abstract category is represented with one AbsCat object. The object includes the name and the type information for the category plus a list of all functions whose return

type is this category. The functions are listed in the order in which they appear in the source code.

type	description
String	the name of the category
[Hypo]	a list of hypotheses
[CatFun]	a list of functions in source-code order

CatFun

This object is used internally to keep a list of abstract functions with their probabilities.

type	description
String	the name of the function
Float	the probability of the function

Type

This is the description of an abstract syntax type. Since the types are monomorphic and in normal form, they have the general form:

$$(x_1 : T_1) \to (x_2 : T_2) \to \ldots \to (x_n : T_n) \to C\ e_1 \ldots e_n$$

The list of hypotheses $(x_i : T_i)$ is stored as a list of Hypo objects and the indices $e_1 \ldots e_n$ are stored as a list of expressions.

type	description
[Hypo]	a list of hypotheses
String	the name of the category in the return type
[Expression]	indices in the return type

Hypo

Every Hypo object represents an argument in some function type. Since we support implicit and explicit arguments, the first field tells us whether we have explicit argument i.e. $(x : T)$ or implicit i.e. $(\{x\} : T)$. The next two fields are the name of the bound variable and its type. If no variable is bound then the name is '_'.

type	description
BindType	the binding type i.e. implicit/explicit argument
String	a variable name or '_' if no variable is bound
Type	the type of the variable

Equation

Every computable function is represented with a list of equations where the equation is a pair of list of patterns and an expression. All equations must have the same number of patterns which is equal to the arity of the function.

type	description
[Pattern]	a sequence of patterns
Expression	an expression

Pattern

This is the representation of a single pattern in a definitional equation for computable function. The first field is a tag which encodes the kind of pattern.

type	description
Int8	a tag

1. tag=0 - pattern matching on constructor application (i.e. $c\, p_1\, p_2\, \ldots\, p_n$)

type	description
String	the name of the constructor
[Pattern]	a list of nested patterns for the arguments

2. tag=1 - a variable

type	description
String	the variable name

3. tag=2 - a pattern which binds a variable but also does nested pattern matching (i.e. $x@p$)

type	description
String	the variable name
Pattern	a nested pattern

4. tag=3 - a wildcard (i.e. $_$).

5. tag=4 - matching a literal i.e. string, integer or float

type	description
Literal	the value of the literal

6. tag=5 - pattern matching on an implicit argument (i.e. $\{p\}$)

type	description
Pattern	the nested pattern

7. tag=6 - an inaccessible pattern ($\sim p$)

type	description
Expr	the nested pattern

Expression

This is the encoding of an abstract syntax expression (tree).

type	description
Int8	a tag

1. tag=0 - a lambda abstraction (i.e. $\backslash x \rightarrow \ldots$)

type	description
BindType	a tag for implicit/explicit argument
String	the variable name
Expression	the body of the lambda abstraction

2. tag=1 - application (i.e. $f\ x$)

type	description
Expression	the left-hand expression
Expression	the right-hand expression

3. tag=2 - a literal value i.e. string, integer or float

type	description
Literal	the value of the literal

4. tag=3 - a metavariable (i.e. $?0, ?1, \ldots$)

type	description
Int	the id of the metavariable

5. tag=4 - an abstract syntax function

type	description
String	the function name

6. tag=5 - a variable

type	description
Int	the de Bruijn index of the variable

7. tag=6 - an expression with a type annotation (i.e. $\langle e : t \rangle$)

type	description
Expression	the annotated expression
Type	the type of the expression

8. tag=7 - an implicit argument (i.e. $\{e\}$)

type	description
Expression	the expression for the argument

Literal

The Literal object represents the built-in kinds of literal constants. It starts with a tag which encodes the type of the constant:

type	description
Int8	literal type

Currently we support only three types of literals:

1. tag=0 - string

type	description
String	the value

2. tag=1 - integer

type	description
Int	the value

3. tag=2 - float

type	description
Float	the value

BindType

The bind type is a tag which encodes whether we have an explicit or an implicit argument. Tag 0 is for explicit, and tag 1 is for implicit.

type	description
Int8	tag

Concrete

Every concrete syntax $\mathcal{C}$ (Definition 3, Section 2.1), in the grammar, is represented with an object. The name of the concrete syntax is the name of the top-level concrete module in the grammar.

type	description
String	the name of the concrete syntax
[Flag]	a list of flags
[PrintName]	a list of print names
[Sequence]	a table with sequences (Section 2.8.1)
[CncFun]	a list of concrete functions
[LinDef]	a list of functions for default linearization
[ProductionSet]	a list of production sets
[CncCat]	a list of concrete categories
Int	the total number of concrete categories allocated for the grammar

Note: The lists Flag, PrintName and CncCat are sorted by name which makes it easy to do binary search.

Note: The total number of concrete categories is used by the parser to determine whether a given category is part of the grammar, i.e. member of $N^{\mathcal{C}}$, or it was created during the parsing. This is the way to decide when to put metavariables during the tree extraction (Section 2.3.7).

PrintName

Every function or category can have a print name which is a user friendly name that can be displayed in the user interface instead of the real one. The print names are defined in the concrete syntax which makes it easier to localize the user interface to different languages.

type	description
String	the name of the function or the category
String	the printable name

Sequence

This is the representation of a single sequence in PMCFG, produced during the common subexpression optimization (Section 2.8.1).

type	description
[Symbol]	a list of symbols

Symbol

The Symbol (Definition 4, Section 2.1) represents either a terminal or a function argument in some sequence. The representation starts with a tag encoding the type of the symbol:

type	description
Int8	expression tag

The supported symbols are:

1. tag=0. This is the representation of an argument, i.e. a pair $\langle k; l\rangle$ where k is the argument index and l is the constituent index.

type	description
Int	argument index
Int	constituent index

2. tag=1 This is again an argument but we use different tag to indicate that the target can be a literal category (see Section 2.6). If the target category is not a new fresh category, generated by the parser, then it is treated as a literal category. In the `pgf_pretty` format, we print this kind of symbols as $\{d; r\}$ instead of $\langle d; r\rangle$.

type	description
Int	argument index
Int	constituent index

3. tag=2 A high-order argument i.e. $\langle d; \$r\rangle$ (Section 2.7).

type	description
Int	argument index
Int	variable number

4. tag=3 This is a terminal symbol and represents a list of tokens.

type	description
[String]	sequence of tokens

5. tag=4 An alternative terminal symbol representing phrase, whose form depends on the prefix of the next token. It corresponds to the **pre** construction in GF and encodes variations like *a/an* in English.

type	description
[String]	the default form
[Alternative]	a sequence of alternatives

Alternative

Every Alternative represents one possible form of a phrase which is dependent on the prefix of the next token. For example when the construction:

pre {"beau"; "bel"/"ami"}

is compiled then the alternative bel / ami will be represented by the pair (["bel"],["ami"]).

type	description
[String]	The tokens to use if the prefix matches
[String]	The prefix matched with the following tokens

CncFun

This is the definition of a single concrete function (Definition 4, Section 2.1). The first field is the name of the corresponding abstract function which gives us the direct definition of the ψ_F mapping. The second field is the function definition given as a list of indices pointing to the sequences table (see the Concrete object).

type	description
String	the name of the corresponding abstract function
[Int]	list of indices into the sequences array

LinDef

The LinDef object stores the list of all concrete functions that can be used for the default linearization of some concrete category (Section 2.5).

type	description
Int	the concrete category
[Int]	a list of concrete functions

ProductionSet

A group of productions with the same result category. The productions are grouped because this makes it easier for the parser to find the relevant productions in the prediction step:

type	description
Int	the result category
[Production]	a list of productions

Production

The production can be either an application of some function or a coercion.

type	description
Int8	tag

1. tag=0 the production is an application (Definition 4, Section 2.1):

type	description
Int	the concrete function
[PArg]	a list of arguments

2. tag=1 the production is a coercion (Section 2.8.1):

type	description
Int	a concrete category

PArg

An argument in a production.

type	description
[Int]	the categories of the high-order arguments (Section 2.7)
Int	a concrete category

CncCat

This is the representation of a set of concrete categories which map to the same abstract category. Since all concrete categories generated from the same abstract category are always represented as consequtive integers, here we store only the first and the last category. The compiler also generates a name for every constituent so here we have the list of names. The length of the list is equal to the dimension of the category.

type	description
String	the name of the corresponding (by ψ_N) abstract category
Int	the first concrete category
Int	the last concrete category
[String]	a list of constituent names

Bibliography

Peter B. Andrews. Theorem proving via general matings. *J. ACM*, 28:193–214, April 1981. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/322248.322249. URL `http://doi.acm.org/10.1145/322248.322249`.

Krasimir Angelov. Incremental parsing with parallel multiple context-free grammars. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics*, EACL '09, pages 69–76, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics. URL `http://dl.acm.org/citation.cfm?id=1609067.1609074`.

Krasimir Angelov and Aarne Ranta. Implementing controlled languages in GF. In *Proceedings of the 2009 conference on Controlled natural language*, CNL'09, pages 82–101, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-14417-9, 978-3-642-14417-2. URL `http://portal.acm.org/citation.cfm?id=1893475.1893482`.

Krasimir Angelov, Björn Bringert, and Aarne Ranta. PGF: A Portable Run-Time Format for Type-Theoretical Grammars. *Journal of Logic, Language and Information*, submitted, 2008.

Wolfgang Bibel. A comparative study of several proof procedures. *Artif. Intell.*, pages 269–293, 1982.

Johan Bos, Stephen Clark, Mark Steedman, James R. Curran, and Julia Hockenmaier. Wide-coverage semantic representations from a CCG parser. In *Proceedings of the 20th International Conference on Computational Linguistics (COLING '04)*, pages 1240–1246, Geneva, Switzerland, 2004.

Pierre Boullier. A proposal for a natural language processing syntactic backbone. Technical Report 3342, INRIA, 1998.

Björn Bringert. Delimited Continuations, Applicative Functors and Natural Language Semantics, 2008. URL `http://www.cse.chalmers.se/alumni/bringert/publ/continuation-semantics/continuation-semantics.pdf`.

Björn Bringert, Krasimir Angelov, and Aarne Ranta. Grammatical framework web service. In *Proceedings of the Demonstrations Session at EACL 2009*, pages 9–12, Athens, Greece, April 2009. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/E09-2003`.

Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *INDAG. MATH*, 34:381–392, 1972.

Håkan Burden and Peter Ljunglöf. Parsing linear context-free rewriting systems. In *Proceedings of the Ninth International Workshop on Parsing Technologies (IWPT)*, pages 11–17, October 2005. URL `citeseer.ist.psu.edu/burden05parsing.html`.

Stephen Clark and James R. Curran. Wide-coverage efficient statistical parsing with CCG and log-linear models. *Comput. Linguist.*, 33:493–552, December 2007. ISSN 0891-2017. doi: http://dx.doi.org/10.1162/coli.2007.33.4.493. URL `http://dx.doi.org/10.1162/coli.2007.33.4.493`.

Michael Collins. Head-driven statistical models for natural language parsing. *Computational Linguistics*, 29:589–637, December 2003. ISSN 0891-2017. doi: http://dx.doi.org/10.1162/089120103322753356. URL `http://dx.doi.org/10.1162/089120103322753356`.

Dana Dannélls. Discourse generation from formal specifications using the Grammatical Framework, GF. *Research in Computing Science*, 46:167–178, 2010. URL `http://spraakdata.gu.se/svedd/pub/vol46-dana.pdf`.

Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, 1970. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/362007.362035.

Ramona Enache and Grégoire Détrez. A framework for multilingual applications on the android platform. In *SLTC 2010 The Third Swedish Language Technology Conference (SLTC 2010)*, pages 37–38, Linköping, 2010.

Michael Hanus. Curry: An integrated functional logic language. Technical report, Curry Community, March 2006.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40:143–184, January 1993. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/138027.138060. URL `http://doi.acm.org/10.1145/138027.138060`.

John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979. ISBN 0-201-02988-X.

Gérard Huet. A unification algorithm for typed lambda-calculus. *Theoretical Computer Science*, 1(1):27–57, 1975. URL `http://linkinghub.elsevier.com/retrieve/pii/0304397575900110`.

Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.34.1287`.

Aravind K. Joshi, Leon S. Levy, and Masako Takahashi. Tree adjunct grammars. *J. Comput. Syst. Sci.*, 10:136–163, February 1975. ISSN 0022-0000. doi: http://dx.doi.org/10.1016/S0022-0000(75)80019-5. URL `http://dx.doi.org/10.1016/S0022-0000(75)80019-5`.

Laura Kallmeyer. *Parsing Beyond Context-Free Grammars*. Springer, 2010. ISBN 978-3-642-14845-3.

Janna Khegai and Aarne Ranta. Multilingual syntax editing in GF. In *Proceedings of the 4th International Conference on Intelligent Text Processing and Computational Linguistics (CICLing 03*, pages 453–464. Springer-Verlag, 2003.

Peter Ljunglöf. *Expressivity and Complexity of the Grammatical Framework*. PhD thesis, Department of Computer Science, Gothenburg University and Chalmers University of Technology, November 2004. URL `http://www.ling.gu.se/~peb/pubs/Ljunglof-2004a.pdf`.

Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of English: the Penn Treebank. *Computational Linguistics*, 19:313–330, June 1993. ISSN 0891-2017. URL `http://portal.acm.org/citation.cfm?id=972470.972475`.

Per Martin-Löf. *Intuitionistic Type Theory*. Napoli: Bibliopolis, 1984.

Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In *Proceedings of the international workshop on Extensions of logic programming*, pages 253–281, New York, NY, USA, 1991. Springer-Verlag New York, Inc. ISBN 3-540-53590-X. URL `http://portal.acm.org/citation.cfm?id=111360.111369`.

Yusuke Miyao and Jun'ichi Tsujii. Probabilistic disambiguation models for wide-coverage HPSG parsing. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, ACL '05, pages 83–90, Stroudsburg, PA, USA, 2005. Association for Computational Linguistics. doi: http://dx.doi.org/10.3115/1219840.1219851. URL http://dx.doi.org/10.3115/1219840.1219851.

Richard Montague. The proper treatment of quantification in ordinary english. In J. Hintikka, J. Moravcsik, and P. Suppes, editors, *Approaches to Natural Language: proceedings of the 1970 Stanford workshop on Grammar and Semantics*, pages 221–242, Dordrecht, 1973. Reidel.

Gopalan Nadathur and Dale Miller. An overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827. MIT Press, 1988.

Gopalan Nadathur and Debra Sue Wilson. A notation for lambda terms I: A generalization of environments. *THEORETICAL COMPUTER SCIENCE*, 198, 1994.

Ryuichi Nakanishi, Keita Takada, and Hiroyuki Seki. An Efficient Recognition Algorithm for Multiple Context-Free Languages. In *Fifth Meeting on Mathematics of Language*. The Association for Mathematics of Language, August 1997. URL http://citeseer.ist.psu.edu/65591.html.

Joakim Nivre. *Inductive Dependency Parsing*, volume 34 of *Text, Speech and Language Technology*. Springer-Verlag, 2006. ISBN 978-1-4020-4888-3. URL http://www.springer.com/education+%26+language/linguistics/book/978-1-4020-4888-3.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

Fernando Pereira and David Warren. *Definite clause grammars for language analysis*, pages 101–124. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1986. ISBN 0-934613-11-7. URL http://dl.acm.org/citation.cfm?id=21922.24330.

Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 199–208, New York, NY, USA, 1988. ACM. ISBN 0-89791-269-1. doi: http://doi.acm.org/10.1145/53990.54010. URL http://doi.acm.org/10.1145/53990.54010.

Frank Pfenning and Carsten Schürmann. System description: Twelf – a meta-logical framework for deductive systems. In *Automated Deduction – CADE-16*, volume 1632 of *Lecture Notes in Computer Science*, pages 679–679. Springer Berlin / Heidelberg, 1999. ISBN 978-3-540-66222-8.

Carl Pollard. *Generalized Phrase Structure Grammars, Head Grammars, and Natural Language*. PhD thesis, Stanford University, CA, 1984.

Carl Jesse Pollard and Ivan A. Sag. *Head-driven phrase structure grammar*. Studies in contemporary linguistics. Center for the Study of Language and Information, 1994. ISBN 9780226674476. URL `http://www.press.uchicago.edu/ucp/books/book/chicago/H/bo3618318.html`.

Xiaochu Qi. An implementation of the language Lambda Prolog organized around higher-order pattern unification. *CoRR*, abs/0911.5203, 2009.

Aarne Ranta. *Type-Theoretical Grammar*. Oxford University Press, 1994. ISBN 9780198538578.

Aarne Ranta. Computational semantics in type theory. *Mathematics and Social Sciences*, 165:31–57, 2004a.

Aarne Ranta. Grammatical Framework: A Type-Theoretical Grammar Formalism. *Journal of Functional Programming*, 14(2):145–189, March 2004b. ISSN 0956-7968. doi: http://dx.doi.org/10.1017/S0956796803004738. URL `http://portal.acm.org/citation.cfm?id=967507`.

Aarne Ranta. Modular Grammar Engineering in GF. *Research on Language & Computation*, 5(2):133–158, June 2007. URL `http://www.springerlink.com/content/r52766j31lg5u075/`.

Aarne Ranta. The GF resource grammar library. *Linguistic Issues in Language Technology*, 2(2), December 2009. URL `http://elanguage.net/journals/index.php/lilt/article/view/214/158`.

Aarne Ranta. *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications, Stanford, 2011. ISBN-10: 1-57586-626-9 (Paper), 1-57586-627-7 (Cloth).

Stefan Riezler, Tracy H. King, Ronald M. Kaplan, Richard Crouch, John T. Maxwell, and Iii Mark Johnson. Parsing the Wall Street Journal using a lexical-functional grammar and discriminative estimation techniques. In *Proceedings of the 40th meeting of the ACL*, pages 271–278, 2002.

Hiroyuki Seki and Yuki Kato. On the Generative Power of Multiple Context-Free Grammars and Macro Grammars. *IEICE-Transactions on Info and Systems*, E91-D(2):209–221, 2008. URL http://ietisy.oxfordjournals.org/cgi/reprint/E91-D/2/209.

Hiroyuki Seki, Takashi Matsumura, Mamoru Fujii, and Tadao Kasami. On multiple context-free grammars. *Theoretical Computer Science*, 88(2):191–229, October 1991. ISSN 0304-3975. URL http://portal.acm.org/citation.cfm?id=123648.

Hiroyuki Seki, Ryuichi Nakanishi, Yuichi Kaji, Sachiko Ando, and Tadao Kasami. Parallel Multiple Context-Free Grammars, Finite-State Translation Systems, and Polynomial-Time Recognizable Subclasses of Lexical-Functional Grammars. In *31st Annual Meeting of the Association for Computational Linguistics*, pages 130–140. Ohio State University, Association for Computational Linguistics, June 1993. URL http://acl.ldc.upenn.edu/P/P93/P93-1018.pdf.

Stuart Shieber, Yves Schabes, and Fernando Pereira. Principles and Implementation of Deductive Parsing. *Journal of Logic Programming*, 24(1&2):3–36, 1995. URL http://citeseer.ist.psu.edu/11717.html.

Mark Steedman. Combinators and Grammars. In Richard Oehrle, Emmon Bach, and Deirdre Wheeler, editors, *Categorial Grammars and Natural Language Structures*, pages 417–442. Reidel, Dordrecht, 1988.

Terrance Swift. Tabling for non-monotonic programming. *Annals of Mathematics and Artificial Intelligence*, 25:201–240, 1999. ISSN 1012-2443. URL http://dx.doi.org/10.1023/A:1018990308362. 10.1023/A:1018990308362.

Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math*, 5:285–309, 1955.

W3C. OWL Web Ontology Language Reference, 2004. URL http://www.w3.org/TR/owl-ref/.