Converting Grammatical Framework to Regulus

Peter Ljunglöf
Department of Linguistics
Göteborg University
Gothenburg, Sweden
peb@ling.gu.se

Abstract

We present an algorithm for converting Grammatical Framework grammars (Ranta, 2004) into the Regulus unification-based framework (Rayner et al., 2006). The main purpose is to take advantage of the Regulus-to-Nuance compiler for generating optimized speech recognition grammars. But there is also a theoretical interest in knowing how similar the two grammar formalisms are.

Since Grammatical Framework is more expressive than Regulus, the resulting Regulus grammars can be overgenerating. We therefore describe a subclass of Grammatical Framework for which the algorithm results in an equivalent Regulus grammar.

1 Background

In this section we describe the grammar formalism Grammatical Framework (GF), and discuss its expressive power and the present options for creating speech recognition grammars (SRGs). The main problem is that the size of the grammar can explode when inflectional parameters are expanded. In this paper we try to solve this problem by converting to a formalism for which there is an optimized SRG compiler. This formalism is Regulus, which is described together with its SRG compiler.

The formal details are left out of the descriptions in this section and can instead be found in section 2. In section 3 the conversion algorithm is presented in detail, and in section 4 there is a short discussion.

1.1 Grammatical Framework

Grammatical Framework (Ranta, 2004) is a grammar formalism based on type theory. The main feature is the separation of abstract and concrete syntax, which makes it very suitable for writing multilingual grammars. A rich module system also facilitates grammar writing as an engineering task, by reusing common grammars.

1.1.1 Separating abstract and concrete syntax

The main idea of GF is the separation of abstract and concrete syntax, a distinction which is shared with several other grammar formalisms such as Abstract Categorial Grammars (de Groote, 2001), Lambda Grammar (Muskens, 2003) and Higher Order Grammar (Pollard, 2004). The abstract part of a grammar defines a set of abstract syntactic structures, called abstract terms or trees; and the concrete part defines a relation between abstract structures and concrete structures.

GF has a *linearization* perspective to grammar writing, where the relation between abstract and concrete is viewed as a mapping from abstract to concrete structures, called linearization terms. In some cases the mapping can be partial or even many-valued.

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

High-level language descriptions: When describing the abstract syntax, the grammar writer can choose not to care about language specific details, such as inflection and word order.

Multilingual grammars: It is possible to define different concrete syntaxes for one particular

Proceedings of SPEECHGRAM 2007, pages 9–16,
Prague, Czech Republic, June 2007.

abstract syntax. Multilingual grammars can be used as a model for interlingua translation, but also to simplify localization of language technology applications.

Resource grammars: The abstract syntax of one grammar can be used as a concrete syntax of another grammar. This makes it possible to implement grammar resources to be used in several different application domains.

These points are currently exploited in the GF Resource Grammar Library (Ranta et al., 2006), which is a multilingual GF grammar with a common abstract syntax for 13 languages. The grammatical coverage is similar to the Core Language Engine (Rayner et al., 2000). The main purpose of the Grammar Library is as a resource for writing domain-specific grammars.

1.1.2 Abstract syntax

The abstract theory of GF is a version of Martin-Löf's (1984) dependent type theory. A grammar consists of declarations of categories and functions. Categories can depend on other categories. Function declarations can bind variables to be used in dependent types, and also take functions as arguments, thus giving rise to higher-order functions. Since the abstract syntax also permits function definitions, the expressive power of GF abstract syntax is Turing-complete.

In this article we restrict ourselves to an important subclass of GF, where there are no dependent types and no higher-order functions. This subclass is called *context-free GF*, and is an instance of Generalized Context-Free Grammar (Pollard, 1984).

The abstract syntax of a context-free GF grammar consists of a set of function typings of the form

$$f \quad : \quad A_1 \to \cdots \to A_\delta \to A$$

This typing says that f is a function taking δ arguments with categories $A_1 \ldots A_\delta$ and returning a category A. This is equivalent to a context-free grammar without terminal symbols. Note however, that the function f would be written $A \to A_1 \ldots A_\delta$ as an ordinary context-free rule. I.e., the left-hand side of a context-free rule corresponds to the result of the function, which is written to the right. The restriction to a context-free backbone is not severe, since the concrete syntax is so expressive.

1.1.3 Concrete syntax

Linearizations are written as terms in a typed functional programming language, which is limited to ensure decidability in generation and in parsing. The language has records and finite-domain functions (called tables); and the basic types are terminal lists (called strings) and finite data types (called parameter types). There are also local definitions, lambda-abstractions and global function definitions. The parameters are declared by the grammar; they can be hierarchical but not recursive, to ensure finiteness.

The language of linearization terms is quite complex, but it can be compiled to a normalized form which is called canonical GF. In this paper we assume that all linearizations are in canonical form. A canonical concrete GF grammar contains declarations of all parameter types, and linearization definitions for all abstract functions.

1.1.4 Expressive power

The expressive power of context-free GF solely depends on the possibility of discontinuous constituents. This means that one grammatical phrase can be split into several parts occurring in different places in the sentence. Discontinuous constituents permit a simple and compositional way of treating, e.g., German compound verbs, where the particle commonly is moved to the end of the sentence.

Ljunglöf (2004) showed that context-free GF is equivalent to Multiple Context-Free Grammar (Seki et al., 1991), which is known to be parseable in time polynomial in the length of the input string. From a converted Multiple CFG, each constituent can be extracted as a context-free rule, which will result in a possibly overgenerating context-free grammar. This context-free grammar can be output from the GF system in several different speech recognition formats, as described by Bringert (2007).

There is a severe problem with the conversion from GF to Multiple CFG however – the size of the resulting grammar tend to explode when the inflectional parameters are expanded. Large grammars such as many of the languages in the Resource

Grammar Library simply cannot be converted. One solution would be to optimize the conversion algorithm, e.g., by interleaving parameter expansion and grammar compaction. Another solution would be to translate into a grammar formalism which does not have this size explosion. This is where Regulus comes in – if we could translate GF grammars into Regulus grammars, we could make use of the research already put into the Regulus-to-Nuance compiler, and would not have to reinvent the wheel.

1.2 Regulus

Regulus is an open-source toolkit for writing grammar-based speech recognition systems (Rayner et al., 2006).[1] The central part is the Regulus grammar formalism and a compiler for creating speech recognition grammars. The toolkit has been enhanced with templates for developing e.g., speech translation systems and dialogue systems. There is also an English resource grammar, which can be used for grammar specialization using explanation-based learning (Rayner et al., 2006, chapters 9–10).

1.2.1 Unification of finite parameters

The Regulus formalism is a context-free grammar, enhanced with unification of finite parameters. This means that the formalism is equivalent to a context-free grammar.

Each context-free category (e.g., Noun) has a number of features (e.g., Number and Gender) with a finite domain of values (e.g., Sg/Pl and Masc/Fem/Neutr). The feature values are specified using a record paired with the grammatical category. Logical variables can be used for unifying features of different constituents in the rule. It is possible to define macros for simplifying common tasks, e.g., when implementing a lexicon.

Compared to Grammatical Framework, the Regulus formalism is quite restricted. There is no clear distinction between abstract and concrete syntax, and there is no advanced module system in which to define grammatical resources. Also, Regulus lacks discontinuous constituents, which reduces the expressive power considerably.

[1]More information about Regulus, including download information, can be found on the project homepage: `http://www.issco.unige.ch/projects/regulus/`

1.2.2 Compiling Regulus to Nuance GSL

Nuance Communications (2003) has developed a context-free grammar format for speech recognition, which has become one of the de facto standards for speech recognition. The grammar format is called Nuance Grammar Specification Language (GSL). The format has some special features, such as semantic tagging and probabilistic grammar rules. There are also restrictions in the format, most notably that the grammars must not be left-recursive.

The Regulus formalism is designed to be able to make use of the special features in Nuance GSL, and the compiler can always create correct GSL grammars without left-recursion.

2 Formal definitions

In this section we give formal definitions of rules and linearization terms in GF, grammar rules and terms in Regulus, and the intermediate structures we will be using.

2.1 GF grammar rules

Since we are only interested in GF grammars with a context-free backbone, the abstract syntax is a context-free grammar where each rule has a unique name. The rules are written as typings in a functional language:

$$f \quad : \quad A_1 \rightarrow \cdots \rightarrow A_\delta \rightarrow A$$

As mentioned earlier, this declaration corresponds to the context-free rule $A \rightarrow A_1 \ldots A_\delta$.

Linearizations are written as function definitions, $f\, x_1 \ldots x_\delta = t$, where $x_1 \ldots x_\delta$ are variables that occur in the linearization term t. An alternative way of writing this is to name the variables consistently for each rule, and then the linearization term t itself is sufficient as a linearization definition. We adopt this idea and use the uniform variable names $\$1 \ldots \$\delta$ in each linearization. With this approach we also distinguish the argument variables from the parameter variables which get bound in tables.

2.2 GF linearization terms and substructures

A parameter type P is just a set of parameter values $\{\mathsf{p}_1, \ldots, \mathsf{p}_n\}$. Note that all parameter types must be disjoint, i.e., each parameter should belong to exactly one parameter type. Linearizations are defined by association linearization terms to the abstract functions. Note that the definition of terms is slightly more general than the definition in GF, since we want to include reduced terms in the definition. The relation between the introduced classes are as follows:

$$\left.\begin{array}{r}\mathbb{P} \subset \mathbb{V}_{\mathbf{Par}} \\ \mathbb{V}_{\mathbf{Str}}\end{array}\right\} \subset \mathbb{V} \subset \mathbb{T}$$

Terms ($t \in \mathbb{T}$) are defined inductively as follows:

$$\begin{array}{rlll} t & ::= & \$n & \text{argument} \\ & \mid & "s" & \text{string} \\ & \mid & t +\!\!+\, t' & \text{concatenation} \\ & \mid & p & \text{pattern} \\ & \mid & \{r_1 = t_1; \ldots; r_n = t_n\} & \text{record} \\ & \mid & t.r & \text{projection} \\ & \mid & [p_1 \Rightarrow t_1; \ldots; p_n \Rightarrow t_n] & \text{table} \\ & \mid & t_1!t_2 & \text{selection} \end{array}$$

where $n > 0$ is a positive integer, $p \in \mathbb{P}$ is a pattern, and $r \in \mathbb{R}$ is a record label. The class $\mathbb{R}$ of record labels is just a finite class of atomic values. The argument reference $\$n$ denotes the nth argument of the linearization function.

Patterns ($p \in \mathbb{P}$) are pairs $x@\pi$ of variables, x, and sets of parameters, $\pi = \{\mathsf{p}_1 \ldots \mathsf{p}_n\}$. The parameters $\mathsf{p}_1 \ldots \mathsf{p}_n$ all must belong to the same parameter type P, i.e., $\pi \subseteq \mathsf{P}$. The meaning of the pattern is that x is bound to one of the parameters in π. If $\pi = \mathsf{P}$ we can skip that part and simply write x. Conversely, if x is not used elsewhere in the term, we can skip it and simply write π.

Note that a pattern is a term in itself, but in GF it will always occur as a single variable x or as a single parameter p. However, after the conversion algorithm has transformed tables into records, patterns will become first class terms.

Reduced terms ($v \in \mathbb{V}$) are subterms of ordinary terms. A reduced term is a term which does not contain a table or a selection, and where projections only occur in the form $\$n.\rho$, where $\rho \in \mathbb{R}^*$ is a sequence of labels:

$$\begin{array}{rll} v & ::= & \$n.\rho \\ & \mid & "s" \\ & \mid & v +\!\!+\, v' \\ & \mid & p \\ & \mid & \{r_1 = v_1; \ldots; r_n = v_n\} \end{array}$$

Reduced parameter terms ($v_p \in \mathbb{V}_{\mathbf{Par}}$) are subterms of reduced terms, which correspond to parameters:

$$v_p \;::=\; \$n.\rho \;\mid\; p$$

Reduced string terms ($v_s \in \mathbb{V}_{\mathbf{Str}}$) are subterms of reduced terms, which correspond to strings:

$$v_s \;::=\; \$n.\rho \;\mid\; "s" \;\mid\; v_s +\!\!+\, v_s'$$

Note that this is equivalent to a sequence of argument projections and string constants, which in turn is equivalent to a right-hand side in a context-free rule.

2.3 Regulus grammar rules and terms

A Regulus grammar is a unification-based phrase-structure grammar. It consists of grammar rules, which in turn is built up using Regulus terms.

Regulus rules are regular context-free grammar rules, where each category is augmented with a Regulus term:

$$A{:}v \;\rightarrow\; \sigma_0 \quad A_1{:}v_1 \quad \sigma_1 \quad \ldots \quad A_\delta{:}v_\delta \quad \sigma_\delta$$

where each A_i is a context-free category, each v_i is a Regulus term, and each σ_i is a (possibly empty) sequence of terminal tokens.

Regulus terms are flat records where all values are patterns ($p_i = x_i@\pi_i$):[2]

$$v_r \;::=\; \{r_1 = p_1; \ldots; r_n = p_n\}$$

In this sense, Regulus terms are just subterms of reduced terms. However, a Regulus term can include one of the two special labels sem and gsem, corresponding to return values and global slot-filling in Nuance GSL, respectively. They can contain complex structures, such as deeply nested lists and records. Using these it is possible to define the syntactical structure of a phrase.

[2]This is a slight abuse of syntax – in Regulus the record row $r_i = x_i@\pi_i$ is written as two separate rows $r_i = x_i; r_i = \pi_i$.

3 Converting GF to Regulus

In this section we describe an algorithm for converting any context-free GF rule into a set of Regulus rules. This conversion is done in three steps:

1. Tables and table selections are removed from the GF term, resulting in several reduced terms and sets of constraints.
2. The results are massaged into Regulus terms for the left-hand side and the right-hand side.
3. Finally, GF abstract syntax is used to create the final Regulus rules.

In the final step, the abstract syntax is added as a semantic value in the Regulus rules. This makes it possible to get back the GF abstract syntax tree, when using Regulus (or Nuance) for parsing the grammar.

Example *As a running example we will use a standard English GF rule for combining transitive verbs with noun phrases:*

$$\begin{array}{lll} \mathsf{vp} & : & \mathsf{TV} \rightarrow \mathsf{NP} \rightarrow \mathsf{VP} \\ & = & \{\mathsf{s} = [n \Rightarrow \$1.\mathsf{s}!n +\!\!+ \$2.\mathsf{s}]\} \end{array}$$

The idea is that a verb phrase has a parametrical number (n), which it inherits from the verb. When the verb phrase is used in a sentence, the number will depend on the inherent number of the subject.

3.1 Converting tables to unification-based records

The main difference between GF and Regulus is that the former has tables and the latter has unification. The problem when converting from GF to Regulus is therefore how to get rid of the tables and selections. We present an algorithm for removing tables and selections, by replacing them with logical variables and equality constraints.

The basic idea is to translate each row $p_i \Rightarrow t_i$ in a table into a record $\{\mathbf{p} = p_i; \mathbf{v} = t_i\}$. The variables in t_i which are bound by p_i become logical variables. Thus the original table gives rise to n different records (if n is the number of table rows), which in the end becomes n different Regulus terms.

A table selection $t!t'$ then has to be translated into the value $t.\mathbf{v}$ of the table, and a constraint is added that the selection term t' and the pattern $t.\mathbf{p}$ of the table should be unified.

Step 1: Removing tables and selections

We define a nondeterministic reduction operation $\Longrightarrow$. The meaning of $t \Longrightarrow v/\Gamma$ is that the term $t \in \mathbb{T}$ is converted to the reduced term $v \in \mathbb{V}$ together with the set of constraints $\Gamma \subseteq \mathbb{V}_{\mathbf{Par}} \times \mathbb{V}_{\mathbf{Par}}$. Each constraint in Γ is of the form $v_p \doteq v'_p$, where v_p and v'_p are reduced parameter terms.

- Each row $p_i \Rightarrow t_i$ in a table is reduced to a record containing the pattern p_i and the reduced value v_i:

  $$\frac{t_i \Longrightarrow v_i/\Gamma_i}{[\ldots; p_i \Rightarrow t_i; \ldots] \Longrightarrow \{\mathbf{p} = p_i; \mathbf{v} = v_i\}/\Gamma_i}$$

  Note that this rule is nondeterministic – the table is reduced to n different terms, where n is the number of table rows.

- A table selection $t!t'$ is reduced to the value $v.\mathsf{v}$, with the added constraint that the pattern $v.\mathsf{p}$ and the selection term v' should unify:

  $$\frac{t \Longrightarrow v_0/\Gamma \qquad t' \Longrightarrow v'_p/\Gamma'}{t!t' \Longrightarrow v \;/\; \Gamma\Gamma'(v_p \doteq v'_p)} \qquad \begin{array}{l} v_p = \mathbf{prj}(v_0, \mathbf{p}) \\ v \;\;= \mathbf{prj}(v_0, \mathbf{v}) \end{array}$$

  Note that since t' denotes a parameter, it will be reduced to a parameter term, $v'_p \in \mathbb{V}_{\mathbf{Par}}$.

- A record projection $t.r$ is reduced to a projection $v.r$:

  $$\frac{t \Longrightarrow v/\Gamma}{t.r \Longrightarrow v_r/\Gamma} \qquad v_r = \mathbf{prj}(v, r)$$

- All other term constructors are reduced compositionally, i.e., by reducing the internal terms recursively.

The function $\mathbf{prj}(v, r)$ calculates the projection of r on the simple term v. Since there are only two reduced term constructors that can correspond to a record, there are only two cases in the definition:

$$\begin{array}{lll} \mathbf{prj}(\{\ldots; r = v_r; \ldots\}, r) & = & v_r \\ \mathbf{prj}(\$n.\rho \qquad\qquad\;\;, r) & = & \$n.\rho r \end{array}$$

Example *The original linearization term of the example contains one table and one selection. The selection* $\$1.\mathsf{s}!n$ *is reduced to* $\$1.\mathsf{s}\mathbf{v}$ *with the added constraint* $\$1.\mathsf{s}\mathbf{p} \doteq n$. *And since there is only one row in the table, the example term is reduced to one single term* v_{vp} *with the constraints* Γ_{vp}*:*

$$\begin{aligned} v_{\mathsf{vp}} &= \{\mathsf{s} = \{\mathbf{p} = n; \mathbf{v} = \$1.\mathsf{s}\mathbf{v} +\!\!+ \$2.\mathsf{s}\}\} \\ \Gamma_{\mathsf{vp}} &= \$1.\mathsf{s}\mathbf{p} \doteq n \end{aligned}$$

3.2 Building Regulus terms

The reduced term and the constraints from the first step do not constitute a Regulus grammar rule, but they have to be massaged into the correct form. This is done in four small steps below.

Step 2a: Flattening the reduced term

After the conversion $t \implies v/\Gamma$, we have a reduced term $v \in \mathbb{V}$ and a set of constraints $\Gamma \in \mathbb{V}_{\mathbf{Par}} \times \mathbb{V}_{\mathbf{Par}}$. Now we convert v into a set of constraints and add to the constraint store Γ:

- For each subterm v' in v denoting a parameter, add $\$0.\rho \doteq v'$ to Γ. The path ρ is the sequence of labels for getting from v to v', i.e., $v' = v.\rho$.

We also create a set of "proto rules" $\Sigma \in \mathbb{R}^* \times \mathbb{V}_{\mathbf{Str}}$:

- For each subterm v' in v denoting a string, add $\rho \to v'$ to Σ. The path ρ is the same as above.

Example *There is one subterm in* v_{vp} *denoting a parameter, so we add* $\$0.\mathsf{s}\mathbf{p} \doteq n$ *to* Γ_{vp}. *There is one subterm in* v_{vp} *that denotes a string, so we let* Σ_{vp} *contain* $\mathsf{s}\mathbf{v} \to \$1.\mathsf{s}\mathbf{v} +\!\!+ \$2.\mathsf{s}$.

Step 2b: Simplifying the constraints

Now we have two constraint stores, Σ and Γ, of which the latter can be partially evaluated into a simpler form.

1. For each constraint of the form $\$n_1.\rho_1 \doteq \$n_2.\rho_2$, we replace it with two new constraints $\$n_i.\rho_i \doteq x$, where x is a fresh variable.[3]

2. For each constraint of the form $x_1@\pi_1 \doteq x_2@\pi_2$, there are two possibilities:
   - If π_1 and π_2 are disjoint, then the constraint is contradictive and we remove v, Γ and Σ from the list of results.
   - Otherwise, we replace each occurrence of $x_i@\pi_i$ in v and in Γ, by $x@\pi$, where x is a fresh variable and $\pi = \pi_1 \cap \pi_2$.

[3]Recall that x is just a shorthand for $x@\pi$, where π is the set of all parameters.

Example *We do not have to do anything to* Γ_{vp}, *since all constraints already are in simplified form.*

Step 2c: Building Regulus terms

Now Γ only contains constraints of the form $\$n.\rho \doteq p$ where $p = x@\pi$. We transform these constraints into the Regulus records $T_0, T_1, \ldots, T_\delta$ in the following way:

- For each constraint $\$n.\rho \doteq p$, add the record row $\{\rho = p\}$ to T_n.

Note that the labels in the Regulus terms are sequences of GF labels.

Example *The constraints in* Γ_{vp} *now give rise to the Regulus terms:*

$$\begin{aligned} T_{\mathsf{vp},0} &= \{\mathsf{s}\mathbf{p} = n\} \\ T_{\mathsf{vp},1} &= \{\mathsf{s}\mathbf{p} = n\} \\ T_{\mathsf{vp},2} &= \{\} \end{aligned}$$

3.3 Building a Regulus grammar

To be able to create Regulus grammar rules from the Regulus terms $T_0 \ldots T_\delta$, we need to look at the abstract syntax in the GF grammar. In this section we will assume that the typing of the linearization in question is $f : A_1 \to \cdots \to A_\delta \to A$.

Regulus (and Nuance GSL) permits the use of arbitrary nested lists in the special sem feature. We will use the nested list for representing a GF abstract syntax tree which then will be returned directly by Nuance after the parsing has succeeded. This is important since the arguments to the function can be permuted in the linearization, which then means that the arguments in the Regulus rule are permuted as well.

Step 3a: Adding GF abstract syntax to the Regulus terms

The abstract syntax tree of the original rule is put as a sem value in the left-hand side term T_0:

- Add the row $\{\mathsf{sem}{=}[f\ x_1 \ldots x_\delta]\}$ to T_0.
- For each $1 \leq i \leq \delta$, add $\{\mathsf{sem}{=}x_i\}$ to T_i.

Note that the $x_1 \ldots x_\delta$ should be fresh variables, not used elsewhere in the terms.

Example *After adding semantics, the example terms become:*

$$
\begin{aligned}
T_{\mathsf{vp},0} &= \{\mathbf{sp} = n; \mathsf{sem} = [\mathsf{vp}\ x\ y]\} \\
T_{\mathsf{vp},1} &= \{\mathbf{sp} = n; \mathsf{sem} = x\} \\
T_{\mathsf{vp},2} &= \{\mathsf{sem} = y\}
\end{aligned}
$$

Step 3b: Building Regulus grammar rules

Finally, we can transform the proto rules in Σ into Regulus grammar rules:

- For each proto rule $\rho_0 \rightarrow v_1 +\!\!+ \cdots +\!\!+ v_m$ in Σ, create a new Regulus rule:

$$A\,\rho_0 : T_0 \quad \rightarrow \quad v_1^\circ\ \ldots\ v_m^\circ$$

where the terms in the right-hand side are calculated as follows:

$$
\begin{aligned}
(\text{"}s\text{"})^\circ &= \text{"}s\text{"} \\
(\$n.\rho)^\circ &= A_n\,\rho : T_n
\end{aligned}
$$

Example *From the single proto rule in* Σ_{vp} *we create the Regulus rule:*

$$\mathsf{VP}_{\mathsf{sv}} : T_{\mathsf{vp},0} \quad \rightarrow \quad \mathsf{TV}_{\mathsf{sv}} : T_{\mathsf{vp},1} \quad \mathsf{NP}_{\mathsf{s}} : T_{\mathsf{vp},2}$$

where the terms $T_{\mathsf{vp},i}$ *are described above.*

4 Discussion

We have presented an algorithm for converting GF grammars with a context-free backbone into unification-based Regulus grammars. The algorithm is simple and straightforward, which is an indication that the formalisms are more similar than one might have guessed beforehand.

4.1 Equivalence of the grammars

The presented algorithm does not necessarily yield an equivalent grammar. This is a consequence of the fact that context-free GF is equivalent to Multiple CFG, and that Multiple CFG is much more expressive than context-free grammars.

However, if the original grammar does not make use of multiple constituents, the conversion is equivalent. Note that the grammar might very well contain multiple constituents, but if there is no right-hand side that refers to both constituents simultaneously the equivalence is still preserved.

4.2 Complexity issues

As mentioned earlier, each GF function might give rise to several Regulus rules, so in one sense the resulting grammar is bigger than the original. However, the actual size in terms of memory usage does not differ (except maybe linear because of differences in syntax).

4.2.1 The number of rules

The number of Regulus rules for one single GF linearization term is equal to:

$$|\Sigma| \prod_{\phi} |\phi|$$

where $|\Sigma|$ is the number of discontinuous constituents, and ϕ ranges over all tables in the linearization term. This is easy to see, since it is only tables that are reduced nondeterministically, and each proto rule in Σ gives rise to one Regulus rule.

4.2.2 The size of the grammar

The total size of the final grammar can be larger than the original GF grammar. This is because the Regulus grammar will contain lots of copies of the same structures, e.g., everything outside of a table has to be duplicated in for each table row. The theoretical limit is the same as above – the number of constituents, times the the total product of all table rows – but in practice the grammar explosion is not that extreme.

Since the increase in size is due to copying, the Regulus grammar can be compacted by the use of macros (Rayner et al., 2006, section 4.5). This could probably be implemented in the algorithm directly, but we have not yet investigated this idea.

4.2.3 Time complexity

The time complexity of the algorithm is approximately equivalent to the size of the resulting Regulus grammar. The first step (in section 3.1), can be implemented as a single pass through the term and is therefore linear in the size of the resulting terms. The post-processing steps (in section 3.2) are also linear in the size of the terms and constraints. Finally, the steps for building grammar rules does not depend on the term size at all. Thus, the time complexity is linear in the size of the final Regulus grammar.

4.3 Using Regulus as a compiler for speech recognition grammars

By presenting an algorithm for converting GF grammars into Regulus, it is possible to further use the Regulus-to-Nuance compiler for getting an optimized speech recognition grammar. The advantage to compiling Nuance grammars directly from GF is clear: the Regulus project has developed and implemented several optimizations (Rayner et al., 2006, chapter 8), which would have to be reimplemented in a direct GF-to-Nuance compiler.

As previously mentioned in section 1.1.4, there is a speech recognition grammar compiler already implemented in GF (Bringert, 2007), which uses the equivalence of GF and Multiple CFG. An interesting future investigation would be to compare the two approaches on realistic grammars.

References

Björn Bringert. 2007. Speech recognition grammar compilation in Grammatical Framework. Submitted to SpeechGram 2007.

Philippe de Groote. 2001. Towards abstract categorial grammars. In *39th Meeting of the Association for Computational Linguistics*, Toulouse, France.

Peter Ljunglöf. 2004. *Expressivity and Complexity of the Grammatical Framework*. Ph.D. thesis, Göteborg University and Chalmers University of Technology, November.

Per Martin-Löf. 1984. *Intuitionistic Type Theory*. Bibliopolis, Napoli.

Reinhard Muskens. 2003. Language, lambdas, and logic. In Geert-Jan Kruijff and Richard Oehrle, editors, *Reosurce Sensitivity in Binding and Anaphora*, pages 23–54. Kluwer.

Nuance Communications, Inc., 2003. *Nuance Speech Recognition System 8.5: Grammar Developer's Guide*, December.

Carl Pollard. 1984. *Generalised Phrase Structure Grammars, Head Grammars and Natural Language*. Ph.D. thesis, Stanford University.

Carl Pollard. 2004. Higher-order categorial grammar. In Michel Moortgat, editor, *International Conference on Categorial Grammars*, Montpellier, France.

Aarne Ranta, Ali El-Dada, and Janna Khegai, 2006. *The GF Resource Grammar Library*. Can be downloaded from the GF homepage `http://www.cs.chalmers.se/~aarne/GF`

Aarne Ranta. 2004. Grammatical Framework, a type-theoretical grammar formalism. *Journal of Functional Programming*, 14(2):145–189.

Manny Rayner, Dave Carter, Pierrette Bouillon, Vassilis Digalakis, and Mats Wirén. 2000. *The Spoken Language Translator*. Cambridge University Press.

Manny Rayner, Beth Ann Hockey, and Pierrette Bouillon. 2006. *Putting Linguistics into Speech Recognition: The Regulus Grammar Compiler*. CSLI Publications.

Hiroyuki Seki, Takashi Matsumara, Mamoru Fujii, and Tadao Kasami. 1991. On multiple context-free grammars. *Theoretical Computer Science*, 88:191–229.