Grammatical Framework and Multiple Context-Free Grammars

Peter Ljunglöf

March 10, 2004

Grammatical Framework (GF) [5] is a grammar formalism originating from logical frameworks for dependent type theory. It is already known that the parsing problem for GF is undecidable, which has to do with the possibility to formulate undecidable propositions. But for subclasses of GF, in particular GF with a context-free backbone, parsing is decidable. Until now the parsing complexity of context-free GF has been unknown, which we aim to change with this article.

We show that there is a simple one-to-one correspondence between Grammatical Framework with context-free backbone and Multiple Context-Free Grammars (MCFG) [7]. Since the parsing complexity for MCFGs is known to be polynomial in the length of the input, we get the same result for context-free GF.

GF with dependent types is undecidable in general, but the separation of abstract and concrete syntax makes it possible to use a two-step process in parsing. First a MCFG parser is used to create a chart, then each item in the chart is converted to a Horn clause. The resulting logic program then solves the parsing problem.

1 Grammatical Framework

Grammatical Framework [5] is a grammar formalism built upon a Logical Framework. What GF adds to the logical framework is a possibility to define concrete syntax, that is, notations expressing formal concepts in user-readable ways. Although GF grammars are bidirectional, the perspective of GF is on generation rather than parsing. A difference from usual grammar formalisms is the support for multilinguality; it is possible to define several concrete syntaxes upon one abstract syntax. The abstract syntax then works as an interlingua between the concrete syntaxes. The development of GF as an authoring system started as a plug-in to the proof editor ALF, to permit natural-language rendering of formal proofs [3]. The extension of the scope outside mathematics was made in the Multilingual Document Authoring project at Xerox [1]. In continued work, GF has been used in areas like software specifications [2] and dialogue systems [6]. In this article we have changed the notation somewhat, compared to other GF articles. The reason for this is to make the comparison with Generalized Context-Free Grammars simpler. There are also some features of GF that we do not mention, mainly syntactic sugar and some default notations.

1.1 GF with a context-free backbone

Grammatical Framework is a strongly typed functional language; functional in the sense that a grammar defines a set of functions, and strongly typed in the sense that each function has a given typing, known at compile-time. Basic types in GF are called *categories*, and are specified by the grammar. For now we assume that there are only a finite number of categories – in section 1.3 we generalize the concept to dependent and higher-order categories.

A function is specified by a typing, an (optional) definition and a linearization. To retain compositionality, all terms of a given category A must linearize to the same *linearization type*, written $[\![A]\!]$.

Function typings The typing of a function tells us how many arguments it takes, what their categories are and what category the result is.

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

The function may take no arguments; in which case it is called a *ground term*. Given a grammar, we define the abstract terms, or syntax trees, as follows. The term $f(t_1, \ldots, t_\delta)$ is of category A whenever each t_i is of category A_i.

Abstract definitions An abstract definition specifies a computation rule on terms, thereby defining a notion of equality between terms. This equality is a semantic equality, since it does not affect the concrete linearizations; even though two terms are equal by definition, they can be linearized to different strings. Since they do not affect the concrete syntax, we do not mention them further in this article.

Linearization terms and types 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 hierarcical but not recursive, to ensure finiteness.

Linearization definitions The linearization of a function term f is a function (also written f) from the linearization types of the argument categories to the linearization type of the result category; or as a functional typing, $f : [\![A_1]\!] \to \ldots \to [\![A_\delta]\!] \to [\![A]\!]$.

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

The linearization ϕ must of course be of type $[\![A]\!]$ (written $\phi : [\![A]\!]$), given that each x_i is of type $[\![A_i]\!]$. Note that a limitation on the linearization is that it is not possible to examine the structure of variables by e.g. case analysis; which is one way of defining compositionality for grammars.

1.2 Canonical GF

The concrete syntax of any GF grammar can be partially evaluated to a grammar in canonical form, as shown in [5]. In canonical form, all local and global definitions disappear, as well as function applications; and all tables are fully expanded. Hierarcical parameters can be flattened; thus we can assume that the parameters are declared by giving a finite set Par of parameter types, each $\mathsf{P} \in \mathsf{Par}$ being a set of parameters $\mathsf{p}_1, \ldots, \mathsf{p}_n$. A linearization term in canonical GF is of the following form:

- A string constant is of type Str; and a concatenation $s_1 \cdot s_2 : \mathsf{Str}$, whenever $s_1, s_2 : \mathsf{Str}$.
- A constant parameter $\mathsf{p} : \mathsf{P}$, whenever $\mathsf{p} \in \mathsf{P}$.
- A record $\{ r_1 = \phi_1 ; \ldots ; r_n = \phi_n \}$ is of type $T = \{ r_1 : T_1 ; \ldots ; r_n : T_n \}$, whenever each $\phi_i : T_i$.
- A record projection $\phi.r_i : T_i$, whenever ϕ is of the record type T above.
- A table $[\mathsf{p}_1 \Rightarrow \phi_1 ; \ldots ; \mathsf{p}_n \Rightarrow \phi_n]$ is of type $\mathsf{P} \Rightarrow T$, whenever $\mathsf{P} = \{ \mathsf{p}_1, \ldots, \mathsf{p}_n \}$, and each $\phi_i : T$.
- A table selection $\phi \,!\, \psi : T$, whenever $\phi : \mathsf{P} \Rightarrow T$ and $\psi : \mathsf{P}$.
- An argument variable $x_i : [\![A_i]\!]$.

Together with this there are computation rules for string concatenation, record projection and table selection.

An example grammar In figure 1, there is a simple example GF grammar, which is not entirely in canonical form. The linearization of q contains a non-expanded table $[n \Rightarrow x.s \,!\, n \cdot y.s]$, whose canonical form is $[\mathsf{Sg} \Rightarrow x.s \,!\, \mathsf{Sg} \cdot y.s ; \mathsf{Pl} \Rightarrow x.s \,!\, \mathsf{Pl} \cdot y.s]$.

$$
\begin{array}{rcl}
[\![S]\!] & = & \{\, s : \mathsf{Str} \,\} \\
[\![N]\!] & = & \{\, s : \mathsf{Str}\,;\ n : \mathsf{Num} \,\} \\
[\![V]\!] & = & \{\, s : \mathsf{Num} \Rightarrow \mathsf{Str} \,\} \\
p & : & N \to V \to S \\
p(x,y) & = & \{\, s = x.s \cdot y.s \,!\, (x.n) \,\} \\
q & : & V \to N \to V \\
q(x,y) & = & \{\, s = [\, n \Rightarrow x.s \,!\, n \cdot y.s \,] \,\} \\
a : N & = & \{\, s = \textit{“animals”};\ n = \mathsf{Pl} \,\} \\
b : N & = & \{\, s = \textit{“Bernie”};\ n = \mathsf{Sg} \,\} \\
l : V & = & \{\, s = [\, \mathsf{Sg} \Rightarrow \textit{“loves”};\ \mathsf{Pl} \Rightarrow \textit{“love”} \,] \,\}
\end{array}
$$

Figure 1: Example GF grammar

1.3 GF with dependent categories

Full GF have a much more expressive abstract syntax than above; categories can depend on other categories. E.g. the grammar could specify that category A depends of category B, meaning that $A(x)$ is a category whenever x is a term of category B.

Another extension is that arguments to function typings (and dependent categories) can be functions and not just of a basic category.

These extensions turn the abstract syntax into a logical framework, where e.g. undecidable propositions can be formulated, thus giving a very expressive formalism. A natural question is then how to parse and linearize terms with these extensions, which will be addressed in section 4.1.

2 Generalized CFG

Generalized Context-Free Grammars (GCFG) were introduced by Pollard in the 80's as a way of formally describing Head Grammars [4]. Later people have used GCFG as a framework for describing many other formalisms, such as Linear Context-Free Rewriting Systems [**?**] and Multiple Context-Free Grammars [7]; and here we will use it to describe GF with a context-free backbone.

There are several definitions of GCFG in the litterature, and we introduce yet another one, in which a Generalized Context-Free Grammar consists of an abstract grammar together with a concrete interpretation.

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

$$A \quad \to \quad f(A_1, \ldots, A_\delta)$$

The tree rewriting relation $A \Rightarrow t$ is defined as $A \Rightarrow f(t_1, \ldots, t_\delta)$ whenever $A_1 \Rightarrow t_1$, ..., $A_\delta \Rightarrow t_\delta$.

Concrete interpretation To each category A is associated a *linearization type* $[\![A]\!]$, which is not further specified. To each function symbol f is associated a partial *linearization function* (also written f), taking as many arguments as the abstract syntax rule specifies.

$$f \quad \in \quad [\![A_1]\!] \to \ldots \to [\![A_\delta]\!] \to [\![A]\!]$$

The linearization of a syntax tree is defined as $[\![f(t_1, \ldots, t_\delta)]\!] = f([\![t_1]\!], \ldots, [\![t_\delta]\!])$, if the application is defined. Note that we impose no restrictions on the linearization types or the linearization functions; this is left to the actual grammar formalism.

2.1 GF with a context-free backbone

Grammatical Framework with a context-free backbone is obviously an instance of GCFG, where the abstract GF rule $f : A_1 \to \ldots \to A_\delta \to A$ is just another way of writing the abstract GCFG rule $A \to f(A_1, \ldots, A_\delta)$.

2.2 Multiple Context-Free Grammars

Multiple Context-Free Grammars [7] were introduced in the late 80's as a very expressive formalism, incorporating Linear Context-Free Rewriting Systems and other mildly context-sensitive formalisms, but still with a polynomial parsing algorithm. MCFG is an instance of GCFG, with the following restrictions on linearizations:

- Linearization types are restricted to tuples of strings.
- The only allowed operations in linearization functions are tuple projections and string concatenations.

Since records can be seen as syntactic sugar for tuples, we can use records in this article without changing the definition of MCFG.

Comparison with GF Obviously MCFG is an instance of context-free GF, but without tables and table selections. The fact that GF can have nested records does not change anything – all nestings can be flattened. Also, an expanded table

$$[\,\mathsf{p}_1 \Rightarrow \phi_1\,;\,\ldots\,;\,\mathsf{p}_n \Rightarrow \phi_n\,] \quad : \quad \mathsf{P} \Rightarrow T$$

is equivalent to a record

$$\{\,\mathsf{p}_1 = \phi_1\,;\,\ldots\,;\,\mathsf{p}_n = \phi_n\,\} \quad : \quad \{\,\mathsf{p}_1 : T\,;\,\ldots\,;\,\mathsf{p}_n : T\,\}$$

and an instantiated selection $\phi\,!\,\mathsf{p}_i$ is equivalent to a record projection $\phi.\mathsf{p}_i$.

There are two fundamental differences:

- MCFG cannot have parameters as linearization values.
- A table selection in GF does not necessarily have to be an instantiated parameter; it can be any term of the correct linearization type.

  In the example grammar, the linearization of p contains a selection $y.s\,!\,(x.n)$, which is not instantiated.

3 Converting GF to MCFG

In this section we show that it is possible to convert a GF grammar into an equivalent grammar where all table selections are instantiated, and containing no parameters. By the argument above, this means that context-free GF and MCFG are equivalent. This conversion is done in two steps, described later in sections 3.2 and 3.3, and the following theorem is a consequence.

Theorem 1 *Any GF grammar with a context-free backbone can be reduced to an equivalent MCFG grammar.*

3.1 Preliminaries

A linearization term ϕ is in *η-normal form* if the structure follows the structure of its linearization type; i.e. ϕ is a record if the type is a record type, and ϕ is an expanded table if the type is a table type. The subterms of ϕ which are of the basic linearization types, Str or $\mathsf{P} \in \mathsf{Par}$, are called the *leaves* of ϕ. A *path* is a sequence of record projections and table selections; meaning that ϵ, $\sigma.r$ and $\sigma\,!\,\phi$ are paths if σ is a path. A path that does not contain any argument variables x_i is called instantiated; in which case the selections ϕ can only be parameters. A non-instantiated path is called nested; this is because if a path contains an argument variable x_i, then that variable is always followed by a (possibly empty) path.

A linearization type T as well as a linearization ϕ can be partitioned into parameter paths and string paths:

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

Note that we equate nestings of tables/records and sets of path-value pairs, and that we extend paths to linearization types in the obvious way. Also note that there are only a finite number of instantiated parameter records $\pi : T^{\mathsf{Par}}$, since there are only finitely many parameters.

The example grammar For the term $a : [\![N]\!]$ in the example grammar we have that

$$\begin{array}{rcl} [\![N]\!]^{\mathsf{Str}} & = & \{\, s : \mathsf{Str} \,\} \\ [\![N]\!]^{\mathsf{Par}} & = & \{\, n : \mathsf{Num} \,\} \\ a^{\mathsf{Str}} & = & \{\, s = \text{“animals”} \} \\ a^{\mathsf{Par}} & = & \{\, n = \mathsf{Pl} \,\} \end{array}$$

3.2 A normal form for linearizations

Definition 1 *A GF linearization is in* table normal form *if it is of the form*

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

and the following hold:

- *ξ contains all parameter paths of the arguments $x_1, \ldots, x_\delta$; in other words $\xi = (x_1^{\mathsf{Par}}, \ldots, x_\delta^{\mathsf{Par}})$.*
- *Each π_k is a possible parameter instantiation of ξ; in other words $\pi_k : [\![A_1]\!]^{\mathsf{Par}} \times \cdots \times [\![A_\delta]\!]^{\mathsf{Par}}$.*
- *Each ϕ_k is in η-normal form where the leaves are either parameters or concatenations of constant strings and instantiated string paths.*

The following algorithm converts any GF linearization in canonical form into normal form.

Algorithm 1 *First, add the outer table as in the definition of table normal form:*

$$\begin{array}{rcl} f(x_1, \ldots, x_\delta) & = & [\, \pi_1 \Rightarrow \phi \,;\, \ldots \,;\, \pi_n \Rightarrow \phi \,] \,!\, \xi \\ \xi & = & (x_1^{\mathsf{Par}}, \ldots, x_\delta^{\mathsf{Par}}) \\ \pi_k & : & [\![A_1]\!]^{\mathsf{Par}} \times \cdots \times [\![A_\delta]\!]^{\mathsf{Par}} \end{array}$$

Second, for each instantiation π_k, convert ϕ to ϕ_k, by repeating the following substitution until there are no parameter paths left:

- *Substitute each instantiated parameter path $x_i.\sigma$ for its π_k-instantiation $(\pi_k)_i.\sigma$.*

Lemma 1 *The algorithm, together with the standard computation rules, yields an equivalent linearization in table normal form.*

The example grammar There are two linearizations in the example that are not in table normal form, and this is how they look after conversion:

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

3.3 Refining the abstract syntax

To get an MCF grammar, we have to get rid of the parameters in some way; and this we do by moving them to the abstract syntax. Each table row $\pi_k \Rightarrow \phi_k$ above will then give rise to a unique function symbol with linearization ϕ_k.

Algorithm 2 *Given a GF grammar where all linearizations are in table normal form, create a grammar with the following categories, function symbols and linearizations:*

- *For each cateogry A and each instantiated parameter record $\pi : [\![A]\!]^{\mathsf{Par}}$, create a new category $\hat{A} = A[\pi]$. The linearization type is the same as the string paths of the original linearization type, $[\![\hat{A}]\!] = [\![A]\!]^{\mathsf{Str}}$*
- *For each syntax rule $A \to f(A_1, \ldots, A_\delta)$, and all new categories $\hat{A}$, $\hat{A}_1$, …, $\hat{A}_\delta$, create a new syntax rule $\hat{A} \to \hat{f}(\hat{A}_1, \ldots, \hat{A}_\delta)$; where $\hat{f}$ is a unique function symbol, $\hat{f} = f[\hat{A} \to \hat{A}_1 \ldots \hat{A}_\delta]$.*
- *For each linearization function*

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

  and each table row $\pi_k \Rightarrow \phi_k$, create a new linearization function for $\hat{f} = f[\hat{A} \to \hat{A}_1 \ldots \hat{A}_\delta]$:

  $$\begin{array}{rcl} \hat{f}(x_1, \ldots, x_\delta) & = & \phi_k^{\mathsf{Str}} \\ \hat{A} & = & A[\phi_k^{\mathsf{Par}}] \\ \hat{A}_i & = & A_i[(\pi_k)_i] \quad (1 \le i \le \delta) \end{array}$$

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

$$
\begin{array}{rcl}
[\![\hat{N}_1]\!] & = & [\![\hat{N}_2]\!] \;=\; [\![\hat{S}]\!] \;=\; \{\, s : \mathsf{Str} \,\} \\
[\![\hat{V}]\!] & = & \{\, s\,!\,\mathsf{Sg} : \mathsf{Str}\,;\; s\,!\,\mathsf{Pl} : \mathsf{Str} \,\} \\
\hat{p}_1 & : & \hat{N}_1 \to \hat{V} \to \hat{S} \\
\hat{p}_1(x,y) & = & \{\, s = x.s \cdot y.s\,!\,\mathsf{Sg} \,\} \\
\hat{p}_2 & : & \hat{N}_2 \to \hat{V} \to \hat{S} \\
\hat{p}_2(x,y) & = & \{\, s = x.s \cdot y.s\,!\,\mathsf{Pl} \,\} \\
\hat{q}_1 & : & \hat{V} \to \hat{N}_1 \to \hat{V} \\
\hat{q}_1(x,y) & = & \{\, s\,!\,\mathsf{Sg} = x.s\,!\,\mathsf{Sg} \cdot y.s\,; \\
 & & \quad s\,!\,\mathsf{Pl} = x.s\,!\,\mathsf{Pl} \cdot y.s \,\} \\
\hat{q}_2 & : & \hat{V} \to \hat{N}_2 \to \hat{V} \\
\hat{q}_2(x,y) & = & \hat{q}_1(x,y) \\
\hat{a} \;:\; \hat{N}_2 & = & \{\, s = \textit{“animals”} \} \\
\hat{b} \;:\; \hat{N}_1 & = & \{\, s = \textit{“Bernie”} \} \\
\hat{l} \;:\; \hat{V} & = & \{\, s\,!\,\mathsf{Sg} = \textit{“loves”};\; s\,!\,\mathsf{Pl} = \textit{“love”} \}
\end{array}
$$

Figure 2: Grammar after conversion to MCFG

Obviously the resulting grammar is an MCF grammar, since all linearizations are records of strings.

Lemma 2 *The resulting grammar is equivalent to the original.*

The example grammar Figure 2 shows how the example grammar looks like after conversion to MCFG.

3.4 Non-deterministic reduction

There is a more direct conversion, using a non-deterministic substitution algorithm. This can also reduce the size of the resulting grammar, when argument parameters are not mentioned in linearizations.

Algorithm 3 *Assume the following abstract syntax rule, together with its linearization function:*

$$
\begin{array}{rcl}
A & \to & f(A_1, \ldots, A_\delta) \\
f(x_1, \ldots, x_\delta) & = & \phi
\end{array}
$$

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

- *Substitute each instantiated parameter path $x_i.\sigma : P$ with any $\mathsf{p} \in P$, such that adding the row $\sigma = \mathsf{p}$ to π_i is consistent.*

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

$$
\begin{array}{rcl}
\hat{A} & \to & \hat{f}(\hat{A}_1, \ldots, \hat{A}_\delta) \\
\hat{f}(x_1, \ldots, x_\delta) & = & \psi^{\mathsf{Str}} \\
\hat{A} & = & A[\psi^{\mathsf{Par}}] \\
\hat{A}_i & = & A_i[\pi_i] \quad (1 \le i \le \delta)
\end{array}
$$

The algorithm is non-deterministic, and we get the final grammar by finding all solutions for each function symbol f; which can be done by a standard all-solutions predicate, such as `findall` in Prolog.

Coercions between categories There is a difference between algorithm 3 and the previous algorithms; if an argument parameter $x_i.\sigma$ is not mentioned in ϕ, then there will be no σ-row in the constraint record π_i. This means that the new category $\hat{A}_i = A_i[\pi_i]$ will only contain a subrecord of $A_i[\phi_i^{\mathsf{Par}}]$, where ϕ_i is a linearization of type $[\![A_i]\!]$.

Algorithm 4 *Given two reduced syntax rules,*

$$
\begin{array}{rcl}
\hat{A} & \to & \hat{f}(\ldots \hat{B}_1 \ldots) \\
\hat{B}_2 & \to & \hat{g}(\ldots)
\end{array}
$$

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

$$
\begin{array}{rcl}
\hat{B}_1 & \to & \hat{c}(\hat{B}_2) \\
\hat{c}(x) & = & x
\end{array}
$$

The example grammar One function symbol gets a different linearization from algorithm 3 than in figure 2; the functions $\hat{q}_1$ and $\hat{q}_2$ get merged into one function $\hat{q}$.

$$\begin{array}{rcl} \hat{q} & : & \hat{V} \to \hat{N} \to \hat{V} \\ \hat{q}(x,y) & = & \{\, s\,!\,\mathsf{Sg} = x.s\,!\,\mathsf{Sg} \cdot y.s\,; \\ & & \quad s\,!\,\mathsf{Pl} = x.s\,!\,\mathsf{Pl} \cdot y.s\,\} \end{array}$$

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

$$\begin{array}{rcl} \hat{N} & \to & \hat{c}_i(\hat{N}_i) \quad (i = 1, 2) \\ \hat{c}_i(x) & = & x \end{array}$$

4 Implications to Parsing

Definition 2 *A* chart *for a GCFG is a finite set of tuples* $(f, \phi, \phi_1, \ldots, \phi_\delta)$*, where* $\phi = f(\phi_1, \ldots, \phi_\delta)$*.*

A tree $t = f(t_1, \ldots, t_\delta)$ *is* represented *by the chart if it contains* $(f, [\![t]\!], [\![t_1]\!], \ldots, [\![t_\delta]\!])$*, and each subtree* t_i *is represented by the chart.*

The following lemma is just another way of saying that GCFG grammars are compositional:

Lemma 3 *The set of GCFG trees* $\{\, t \mid [\![t]\!] = \phi \,\}$*, for a given* ϕ*, can be represented by a single chart.*

In other words, a correct parsing algorithm for GCFGs does not have to return anything more than a chart.

If we have translated a context-free GF grammar into MCFG using algorithm 1+2, it is straight-forward to translate back a chart for the MCFG into a chart for the original grammar. Each item $(\hat{f}, \phi, \phi_1, \ldots, \phi_\delta)$, where $\hat{f} = f[\hat{A} \to \hat{A}_1 \ldots \hat{A}_\delta]$ and $\hat{A}_i = A_i[\pi_i]$, can be converted to the item $(f, \phi \cup \pi, \phi_1 \cup \pi_1, \ldots, \phi_\delta \cup \pi_\delta)$. Back-translation of trees are even simpler; just strip off the extra information from the nodes – each tree node $\hat{f} = f[\ldots]$ is converted to f.

If we have converted using algorithm 3+4, back-translation is only slightly more complicated; if there is a coercion $\hat{A}_k \to \hat{c}(\hat{A}_k)$, where $\hat{A}'_k = A_k[\pi'_k]$, use the linearization $\phi_k \cup \pi'_k$ instead of $\phi_k \cup \pi_k$.

4.1 Dependent categories

If we have a GF grammar with dependent categories, there is a straight-forward two-step parsing process for that grammar. First we simply remove all dependencies form the abstract syntax, thereby getting a grammar with a context-free backbone. This grammar is over-generating, so when parsing we get a chart containing all parse trees we want, but perhaps also some unwanted trees.

The second step is to convert the chart into Horn clauses, which can be solved by any proof search, e.g. standard Prolog. This conversion is done one item at the time; suppose the following chart item:

$$(f, \phi, \phi_1, \ldots, \phi_\delta)$$

where f has the following abstract typing:

$$\begin{array}{rcl} f & : & (x_1 : A_1) \to \ldots \to (x_\delta : A_\delta(x_1, \ldots, x_{\delta-1})) \\ & & \quad \to A(x_1, \ldots, x_\delta) \end{array}$$

From this we can create the following Horn clause (where $t : A[\phi]$ is just syntactic sugar for a 3-tuple):

$$\begin{array}{l} f(x_1, \ldots, x_\delta) : A(x_1, \ldots, x_\delta)[\phi] \ :- \\ \quad x_1 : A_1[\phi_1],\ \ldots,\ x_\delta : A_\delta(x_1, \ldots, x_{\delta-1})[\phi_\delta] \end{array}$$

Finally, the query $:- x : S[\phi]$, where x is a logic variable, will result in all possible parse trees x of category S, linearizing to the input string ϕ.

4.2 Functional categories

In full GF, arguments to functions can themselves be functions. This gives rise to the question of how to linearize an "incomplete" category $B_1 \to \ldots \to B_\delta \to B$. This is solved in GF by pairing the linearization of the result category B with linearizations of the *variable bindings* representing objects of category B_1, $\ldots$, B_δ.

Formally, each occurence of a function category $B_1 \to \ldots \to B_\delta \to B$ as an argument in a typing is replaced by the new category $\hat{B}$, with linearization type

$$[\![\hat{B}]\!] \quad = \quad [\![B]\!] \times [\![\mathsf{Var}]\!]^\delta$$

where Var is a unique category for recognizing variable bindings, specified by the grammar. In GF, the default linearization type of variables is $[\![\mathsf{Var}]\!] = \mathsf{Str}$, and it recognizes strings looking like ordinary mathematical variables (*"x"*, *"y"*, *"z"*, ...).

For each new category $\hat{B}$ we also need a coercion $\hat{b}$:

$$\begin{aligned} \hat{b} &: B \to \mathsf{Var} \to \ldots \to \mathsf{Var} \to \hat{B} \\ \hat{b}(x, y_1, \ldots, y_\delta) &= (x, y_1, \ldots, y_\delta) \end{aligned}$$

This conversion show that adding function arguments to abstract typings does not change the expressive power of GF, and that they are possible to handle with the parsing algorithms described in this paper.

References

[1] Marc Dymetman, Veronica Lux, and Aarne Ranta. XML and multilingual document authoring: Convergent trends. In *COLING*, pages 243–249, Saarbrücken, Germany, 2000.

[2] Reiner Hähnle, Kristofer Johannisson, and Aarne Ranta. An authoring tool for informal and formal requirements specifications. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering*, volume 2306 of *LNCS*, pages 233–248. Springer, 2002.

[3] Thomas Hallgren and Aarne Ranta. An extensible proof text editor. In M. Parigot and A. Voronkov, editors, *LPAR-2000*, volume 1955 of *LNCS/LNAI*, pages 70–84. Springer, 2000.

[4] Carl Pollard. *Generalised Phrase Structure Grammars, Head Grammars and Natural Language*. PhD thesis, Stanford University, 1984.

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

[6] Aarne Ranta and Robin Cooper. Dialogue systems as proof editors. *Journal of Logic, Language and Information*, 13(2):225–240, April 2004.

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