Noname manuscript No.
(will be inserted by the editor)

PGF: A Portable Run-Time Format for Type-Theoretical Grammars

Krasimir Angelov · Björn Bringert · Aarne Ranta

Received: date / Accepted: date

Abstract PGF (Portable Grammar Format) is a low-level language used as a target of compiling grammars written in GF (Grammatical Framework). Low-level and simple, PGF is easy to reason about, so that its language-theoretic properties can be established. It is also easy to write interpreters that perform parsing and generation with PGF grammars, and compilers converting PGF to other format. This paper gives a concise description of PGF, covering syntax, semantics, and parser generation. It also discusses the technique of embedded grammars, where language processing tasks defined by PGF grammars are integrated in larger systems.

1 Introduction

PGF (Portable Grammar Format) is a grammar formalism designed to capture the computational core of type-theoretical grammars, such as those written in GF (Grammatical Framework, Ranta 2004). PGF thus relates to GF in the same way as JVM (Java Virtual Machine) bytecode relates to Java. While GF (like Java) is a rich language, whose features help the programmer to express her ideas on a high level of abstraction, PGF (like JVM) is an austere language, which is easy to implement on a computer and easy to reason about. The bridge between these two level, in both cases, is a compiler. The compiler gives the grammar writer the best of the two worlds: when writing grammars, she can concentrate on linguistic ideas and find concise expressions for them; when using grammars, she can enjoy efficient run-time performance and a light-weight run-time system, as well as integration in applications.

PGF was originally designed as a back-end language for GF, providing the minimum of what is needed to perform generation and parsing. Its expressive power is between context-free and fully context-sensitive (Ljunglöf 2004). Thus it can potentially provide a back end to numerous other grammar formalisms as well, providing for free a large part of what is

Krasimir Angelov · Björn Bringert · Aarne Ranta
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg, Sweden
E-mail: {krasimir,bringert,aarne}@chalmers.se

involved in implementing these formalisms. In addition, PGF can be compiled further into other formats, such as language models for speech recognition systems (Bringert 2007a).

The most prominent characteristic of PGF (as well as GF) is **multilinguality**: a PGF grammar has an **abstract syntax**, which is a type-theoretical definition of tree structures. The abstract syntax is equipped with a set of **concrete syntaxes**, which are reversible mappings of tree structures to strings. This grammar architecture is known as the Curry architecture, with a reference to Curry (1961). It is at the same time familiar from programming language descriptions, dating back to McCarthy (1962). While the Curry architecture was used by Montague (1974) for describing English, GF might be the first grammar formalism that exploits the multilingual potential of the Curry architecture.

Multilingual PGF grammars readily support translation and multilingual generation. They are also useful when writing monolingual applications, since the non-grammar parts of an application typically communicate with the abstract syntax alone. This follows the standard architecture of compilers, which use an abstract syntax as the format in which a programming language is presented to the other components, such as the type checker and the code generator (Appel 1997). Thus an application using PGF is easy to port from one target language to another, by just writing a new concrete syntax. The GF **resource grammar library** (Ranta 2008) helps application programmers by providing a comprehensive implementation of syntactic and morphological rules for 15 languages.

JVM is a general-purpose Turing-complete language, but PGF is limited to expressing grammars. Grammars can be seen as declarative programs performing tasks such as parsing and generation. In a language processing system, these tasks usually have to be combined with other functionalities. For instance, a question answering system reads input and produces output by using grammars, but the program that computes the answers from the questions has to be written in another language. **Embedded grammars** is a technique that enables programs written in another programming language to call PGF functionalities and also to manipulate PGF syntax trees as data objects. There are two main ways to implement this idea: interpreters and compilers. A PGF interpreter is a program written in a general-purpose language (such as C++, Haskell, or Java, which we have already written interpreters for). The interpreter reads a PGF grammar from a file and gives access to parsing and generation with the grammar. A PGF compiler translates PGF losslessly to another language (such as C, JavaScript, and Prolog, for which compilers have already been written). When we say that PGF is portable, we mean that one can write PGF interpreters and compilers for different host languages, so that the same PGF grammars can be used as components in applications written in these host languages.

The purpose of this paper is to describe the PGF grammar format and briefly show how it is used in applications. The PGF description is detailed enough to enable the reader to write her own PGF interpreters and compilers; but it is informal in the sense that we don't fully specify the concrete format produced by the current GF-to-PGF compiler and implemented by the current PGF interpreters. For that level of detail, on-line documentation is more appropriate and can be found on the GF web page[1].

Section 2 gives a concise but complete description of the syntax and semantics of PGF. Section 3 gives a summary of the expressive power of PGF, together with examples illustrating its main properties. It also discusses some extensions of PGF. Section 4 describes the parser generation and sketches the parsing algorithm of PGF. Section 5 discusses the principal ways of using PGF in language processing applications, the compilation from PGF to other formats, and the compilation of PGF grammars from GF, which is so far the main

[1] http://gf.digitalgrammars.com/

way to produce PGF grammars. Section 6 gives a survey of actual applications running PGF and provides some data evaluating its performance. Section 7 discusses related work, and Section 8 gives a conclusion.

2 The syntax and semantics of PGF

2.1 Multilingual grammar

The top-most program unit of PGF is a **multilingual grammar** (briefly, grammar). A grammar is a pair of an **abstract syntax** $\mathscr{A}$ and a set of **concrete syntaxes** $\mathscr{C}_1, \ldots, \mathscr{C}_n$, as follows:

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

2.2 Abstract syntax

An abstract syntax has a finite a set of **categories** and a finite set of **functions**. Categories are defined simply as unique identifiers, whereas functions are unique identifiers equipped with **types**. A type has the form

$$(\mathsf{C}_1, \ldots, \mathsf{C}_n) \to \mathsf{C}$$

where n may be 0 and each of $\mathsf{C}_1, \ldots, \mathsf{C}_n, \mathsf{C}$ is a category. These types are actually **function types** with **argument types** $\mathsf{C}_1, \ldots, \mathsf{C}_n$ and **value type** C.

Here is an example of an abstract syntax, where we use the keyword **cat** to give the categories and **fun** to give the functions. It defines the categories S (sentence), NP (noun phrase), and VP (verb phrase). Sentences are formed by the Pred function. John is given as an example of a noun phrase and Walk of a verb phrase.

cat S; NP; VP
fun Pred : (NP, VP) → S
fun John : () → NP
fun Walk : () → VP

An **abstract syntax tree** (briefly, tree) is formed by applying the functions obeying their typings, as in simply typed lambda calculus. Thus, for instance, Pred (John, Walk), is a tree of type S, usable for representing the string *John walks*.

2.3 Concrete syntax

A concrete syntax has judgements assigning a **linearization type** to each category in the abstract syntax and a **linearization function** to each function. Linearization types are **tuples** built from **strings** and **bounded integers**. Thus we have the following forms of linearization types T:

$$\mathsf{T} ::= \mathsf{Str} \mid \mathsf{Int}_n \mid \mathsf{T}_1 * \ldots * \mathsf{T}_n$$

Linearization functions are **terms** t of these types, built in the following ways:

$$t ::= [\,] \mid \text{“foo”} \mid t_1 ++ t_2 \mid i \mid <t_1, \ldots, t_n> \mid t_1 \,!\, t_2 \mid \$_i$$

Strings:

$$[\,]:\mathsf{Str} \quad \text{“foo”}:\mathsf{Str} \quad \frac{s,t:\mathsf{Str}}{s ++ t:\mathsf{Str}}$$

Bounded integers:

$$i:\mathsf{Int}_i \quad \frac{i:\mathsf{Int}_m}{i:\mathsf{Int}_n}\ m<n$$

Tuples:

$$\frac{t_1\ :\ T_1\ \ldots\ t_n\ :\ T_n}{<t_1,\ldots,t_n>:\ T_1 * \cdots * T_n}$$

Projections:

$$\frac{t:\mathsf{T}^n\ \ u:\mathsf{Int}_n}{t\,!\,u:\mathsf{T}} \quad \frac{t:\mathsf{T}_1 * \ldots * \mathsf{T}_n}{t\,!\,i:\mathsf{T}_i}\ i=1,\ldots,n$$

Argument variables:

$$\frac{}{T_1,\ldots,T_n \vdash \$_i:\mathsf{T}_i}\ i=1,\ldots,n$$

Table 1 The type system of PGF concrete syntax.

The first three forms are canonical for strings: the empty string $[\,]$, a token (quoted, like “foo”), and the **concatenation** $++$. Concatenation is associative, and it preserves the separation between tokens. The empty string is nilpotent with respect to concatenation. The canonical PGF representation of the string *John loves Mary*, if conceived as consisting of three tokens, is thus “John” $++$ “loves” $++$ “Mary”.

The form i, where i is a numeric constant $1,2,\ldots,n$, is canonical for integers bounded by n. The form $<t_1,\ldots,t_n>$ is canonical for tuples, comprising a term t_i for each component type T_i in a tuple type.

The last two forms in the term syntax are non-canonical. The form $t_1\,!\,t_2$ is the **projection** of t_2 from t_1. In order for the projection to be well-formed, t_1 must be a tuple and t_2 an integer within the bounds of the size of the tuple. The last form $\$_i$ is an **argument variable**. It is bound to a term given in a context, which, as we shall see, always consists of the linearizations of the subtrees of a tree.

Table 1 gives the static typing rules of the terms in PGF. It defines the relation $\Gamma \vdash t\ :\ T$ (“in context Γ, term t has type T”). The context is a sequence of types, and it is left implicit in all rules except the one for argument variables.

The context is in each linearization function created from the linearization types of the arguments of the function. Thus, if we have

fun $f:(\mathsf{C}_1,\ldots,\mathsf{C}_n)\to\mathsf{C}$
lincat $\mathsf{C}_1=\mathsf{T}_1;\ldots;\mathsf{C}_n=\mathsf{T}_n$
lincat $\mathsf{C}=\mathsf{T}$
lin $f=t$

then we must also have

$$T_1,\ldots,T_n \vdash t\ :\ T$$

The alert reader may notice that the typing rules for projections are partial: they cover only the special cases where either all types in the tuple are the same (denoted T^n) or the index is an integer constant. If the types are different and the index has an unknown value (which happens when it e.g. depends on an argument variable), the type of the projection cannot be known at compile time. As we will see in Section 5.4, PGF grammars compiled from GF always fall under these special cases.

2.4 Examples of a concrete syntax

Let us build a concrete syntax for the abstract syntax of the Section 2.2. We define the linearization type of sentences to be just strings, whereas noun phrases and verb phrases are more complex, to account for **agreement**. Thus a noun phrase is a pair of a string and an agreement feature, where the feature is represented by an integer. A verb phrase is a tuple of strings—as many as there are possible agreement features. In this simple example, we just assume two features, corresponding to the singular and the plural.

$$\textbf{lincat}\ \mathsf{S} = \mathsf{Str}; \mathsf{NP} = \mathsf{Str} * \mathsf{Int}_2; \mathsf{VP} = \mathsf{Str} * \mathsf{Str};$$

The linearization of John is the string "John" with the feature 1, representing the singular. The linearization of Walk gives the two forms of the verb *walk*.

$$\textbf{lin}\ \mathsf{John} = <\text{"John"}, 1>; \mathsf{Walk} = <\text{"walks"}, \text{"walk"}>$$

The agreement itself is expressed as follows: in predication, the first field (1) of the noun phrase ($\$_1$)—is concatenated with a field of the verb phrase ($\$_2$). The field that is selected is given by the second field of the first argument ($\$_1\,!\,2$):

$$\textbf{lin}\ \mathsf{Pred} = \$_1\,!\,1 +\!\!+ \$_2\,!\,(\$_1\,!\,2)$$

The power of a multilingual grammar comes from the fact that both linearization types and linearization functions are defined independently in each concrete syntax. Thus German, in which both a two-value number and a three-value person are needed in predication can be given the following concrete syntax:

$$\begin{array}{l}\textbf{lincat}\ \mathsf{NP} = \mathsf{Str} * \mathsf{Int}_2 * \mathsf{Int}_3\\ \qquad\quad \mathsf{VP} = (\mathsf{Str} * \mathsf{Str} * \mathsf{Str}) * (\mathsf{Str} * \mathsf{Str} * \mathsf{Str})\\ \textbf{lin}\ \mathsf{Pred} = \$_1\,!\,1 +\!\!+ (\$_2\,!\,(\$_1\,!\,2))\,!\,(\$_1\,!\,3)\\ \quad \mathsf{John} = <\text{"John"}, 1, 3>\\ \quad \mathsf{Walk} = <<\text{"gehe"}, \text{"gehst"}, \text{"geht"}>, <\text{"gehen"}, \text{"geht"}, \text{"gehen"}>>\end{array}$$

2.5 Linearization

Linearization—the operation $\mapsto$ converting trees into concrete syntax objects—is performed by following the operational semantics given in Table 2. The table defines the relation $\gamma \vdash t \Downarrow v$ ("in context γ, term t evaluates to term v"). The context is a sequence of terms, and it is left implicit in all rules except the one for argument variables.

To linearize a tree, we linearize its immediate subtrees, and the sequence of the resulting terms constitutes the context of evaluation for the full tree:

$$\frac{a_1 \mapsto t_1 \ \ldots\ a_n \mapsto t_n \quad t_1, \ldots, t_n \vdash t \Downarrow v}{f\, a_1 \ldots a_n \mapsto v}\ \textbf{lin}\, f = t$$

Since linearization operates on the linearizations of immediate subtrees, it is a homomorphism, in other words, a compositional mapping. A crucial property of PGF making it possible to maintain the compositionality of linearization in realistic grammars is that its value need not be a string, but a richer data structure. If strings are what ultimately are needed, one can require that the start category has the linearization type Str; alternatively, one can define a realization operation that finds the first string in a tuple recursively.

Strings:

$$[] \Downarrow [] \quad \text{“foo”} \Downarrow \text{“foo”} \quad \frac{s \Downarrow v \;\; t \Downarrow w}{s ++ t \Downarrow v ++ w}$$

Bounded integers:

$$i \Downarrow i$$

Tuples:

$$\frac{t_1 \Downarrow v_1 \;\ldots\; t_n \Downarrow v_n}{<t_1,\ldots,t_n> \Downarrow <v_1,\ldots,v_n>}$$

Projections:

$$\frac{t \Downarrow <v_1,\ldots,v_n> \;\; u \Downarrow i}{t!u \Downarrow v_i}\; i = 1,\ldots,n$$

Argument variable:

$$v_1,\ldots,v_n \vdash \$_i \Downarrow v_i \; (i = 1,\ldots,n)$$

Table 2 The operational semantics of PGF.

3 Properties of PGF

3.1 Expressive power

Context-free grammars have a straightforward representation in PGF: consider a rule

$$C \longrightarrow t_1 \ldots t_n$$

where every t_i is either a nonterminal C_j or a terminal s. This rule can be translated to a pair of an abstract function and a linearization:

fun $f : (\mathsf{C}_1,\ldots,\mathsf{C}_m) \to \mathsf{C}$
lin $f = u_1 ++ \ldots ++ u_n$

where f is a unique label, $(\mathsf{C}_1,\ldots,\mathsf{C}_m)$ are the nonterminals in the rule, and each u_i is either $\$_j$ (if $t_i = \mathsf{C}_j$) or s (if $t_i = s$).

That PGF is stronger than context-free grammars, is easily shown by the language $a^n b^n c^n$, whose grammar shows how **discontinuous constituents** are encoded as tuples of strings. The grammar is the following:

cat $\mathsf{S}; \mathsf{A}$
fun $s : (\mathsf{A}) \to \mathsf{S}; e : () \to \mathsf{A}; a : (\mathsf{A}) \to \mathsf{A}$
lincat $\mathsf{S} = \mathsf{Str}; \mathsf{A} = \mathsf{Str} * \mathsf{Str} * \mathsf{Str}$
lin $s = \$_1 ! 1 ++ \$_1 ! 2 ++ \$_1 ! 3$
$e = <[],[],[]>$
$a = <\text{“a”} ++ \$_1 ! 1, \text{“b”} ++ \$_1 ! 2, \text{“c”} ++ \$_1 ! 3>$

The general result about PGF is that it is equivalent to PMCFG (Parallel Multiple Context-Free Grammar, Seki et al. 1991). Hence any PGF grammar is parsable in polynomial time, with the exponent dependent on the grammar. This result was obtained for a more complex subset of GF by Ljunglöf (2004). Section 4 on parser generation will outline the result for PGF; it will also present a definition of PMCFG.

Being polynomial, PGF is not fully context-sensitive, but it is not mildly context-sensitive in the sense of (Joshi et al. 1991), because it does not have the **constant-growth property**. A counterexample is the following grammar, which defines the exponential language $\{a^{2^n} \mid n = 0,1,2,\ldots\}$:

cat S
fun $a : () \to \mathsf{S}; s : (\mathsf{S}) \to \mathsf{S}$
lincat S = Str
lin $a =$ "a"; $s = \$_1 ++ \$_1$

3.2 Extensions of concrete syntax

A useful extension of concrete syntax is **free variation**, expressed by the operator $|$. Free variation is used in concrete syntax to indicate that different expressions make no semantic difference, that is, that they have the same abstract syntax. A term $t \mid u$ is well-formed in any type T, if both t and u have type T. In operational semantics, free variation requires the lifting of other operations to cover operands of the form $t|u$. For instance,

$$(t \mid u) ! v = (t ! v) \mid (u ! v)$$

As shown by Ljunglöf (2004), the semantics can be given in such a way that the complexity of parsing PGF is not affected.

In practical applications of PGF, it is useful to have non-canonical forms that decrease the size of grammars by factoring out common parts. In particular, the use of **macros** factors out terms occurring in several linearization rules. The method of **common subexpression elimination** often results in a grammar whose code size is just one tenth of the original. This method is standardly applied as a back-end optimization in the GF grammar compiler (Section 5.4), which may otherwise result in code explosion.

3.3 Extensions of abstract syntax

One of the original motivations of GF was to implement type-theoretical semantics as presented in Ranta (1994). This kind of semantics requires that the abstract syntax notation has the strength of a **logical framework**, in the sense of Martin-Löf (1984) and Harper et al. (1993). What we have presented above is a special case of this, with three ingredients missing:

- **Dependent types**: a category may take trees as arguments.
- **Higher-order abstract syntax**: a function may take functions as arguments.
- **Semantic definitions**: trees may be given computation rules.

Ranta (2004) gives the typing rules and operational semantics for these extensions of GF. The extensions have been transferred to PGF as well, but only the Haskell implementation currently supports them. The reason for this is mainly that these extensions are rarely used in GF applications (Burke and Johannisson 2005 and Jonson 2006 being exceptions). Language-theoretically, their effect is either dramatic or null, depending on how the language defined by a grammar is conceived. If parsing is expected to return only well-typed trees, then dependent types together with semantic definitions makes parsing undecidable, because type checking is undecidable. If parsing and dependent type checking are separated, as they are in practical implementations, dependent types make no difference to the parsing complexity.

Metavariables are an extension useful for several purposes. In particular, they are used for encoding suppressed arguments when parsing grammars with erasing linearization rules

(Section 4.5). In dependent type checking, they are used for unifying type dependencies, and in interactive syntax editors (Khegai et al. 2003), they are used for marking unfinished parts of syntax trees.

4 Parsing

PGF concrete syntax is simple but still too complex to be used directly for efficient parsing and for that reason it is converted to PMCFG. It is possible to do the conversion incrementally during the parsing but this slows down the process, so instead we do the conversion in the compiler. This means that we have duplicated information in the PGF file, because the two formalisms are equivalent, but this is a trade off between efficiency and grammar size. At the same time it is not feasible to use only the PMCFG because it might be quite big in some cases while the client might be interested only in linearization for which he or she can use only the PGF syntax.

4.1 PMCFG definition

A parallel multiple context-free grammar is a 8-tuple $G = (N, T, F, P, S, d, r, a)$ where:

- N is a finite set of categories and a positive integer $d(A)$ called dimension is given for each $A \in N$.
- T is a finite set of terminal symbols which is disjoint with N.
- F is a finite set of functions where the arity $a(f)$ and the dimensions $r(f)$ and $d_i(f)$ $(1 \leq i \leq a(f))$ are given for every $f \in F$. Let's for every positive integer d, $(T^*)^d$ denote the set of all d-tuples of strings over T. The function 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 it is defined as:
  $f :=< \alpha_1, \alpha_2, \ldots, \alpha_{r(f)} >$
  Each α_i is a string 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.
- P is a finite set of productions of the form:
  $A \rightarrow f[A_1, A_2, \ldots, A_{a(f)}]$
  where $A \in N$ is called result category, $A_1, A_2, \ldots, A_{a(f)} \in N$ are called argument categories and $f \in F$ is the 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.
- S is the start category and $d(S) = 1$.

We use the same definition of PMCFG as is used by Seki and Kato (2008) and Seki et al. (1993) with the minor difference that they use variable names like x_{kl} while we use $\langle k; l \rangle$ to refer to the function arguments. In the actual implementation we also allow every category to be used as a start category. When the category has multiple constituents then they all are tried from the parser.

The $a^n b^n c^n$ language from section 3.1 is represented in PMCFG as shown in Figure 1. The dimension of S and A are $d(S) = 1$ and $d(A) = 3$. Functions s and e are with 0-arity and function a is with 1-arity i.e. $a(s) = 0$, $a(a) = 1$ and $a(e) = 0$.

$$S \to s[A]$$
$$A \to a[A]$$
$$A \to e[]$$

$$s := < \langle 1;1\rangle\ \langle 1;2\rangle\ \langle 1;3\rangle >$$
$$a := < a\ \langle 1;1\rangle, b\ \langle 1;2\rangle, c\ \langle 1;3\rangle >$$
$$e := < [], [], [] >$$

Fig. 1 PMCFG for the $a^n b^n c^n$ language.

4.2 PMCFG generation

Both PGF and PMCFG deal with tuples but PGF is allowed to have bounded integers and nested tuples while PMCFG is restricted to have only flat tuples containing only strings. To do the conversion we need to get rid of the integers and to flatten the tuples.

Let's take again the categories from section 2.4 as examples:

lincat $\mathsf{NP} = \mathsf{Str} * \mathsf{Int}_2 * \mathsf{Int}_3$
$\mathsf{VP} = (\mathsf{Str} * \mathsf{Str} * \mathsf{Str}) * (\mathsf{Str} * \mathsf{Str} * \mathsf{Str})$

The type for VP does not contain any integers so it is simply flattened to one tuple with six constituents. When the linearization type contains integers then the category is split into multiple PMCFG categories, one for each combination of integer values. The integers are removed and the remaining type is flattened to one tuple. In this case for NP we will have six categories and they will all have a single string as a linearization type. Category splitting is not an uncommon operation. For example, in part-of-speech tagging there are usually different tags for nouns in plural and nouns in singular. We do this also on a syntactic level and the NP category is split into:

NP_{11} NP_{12} NP_{13}
NP_{21} NP_{22} NP_{23}

The conversion from PGF rules to PMCFG productions starts with abstract interpretation (Table 3), which is very similar to the operational semantics in Table 2. The major difference is that the argument values are known only at run-time and actually the $\$_i$ terms should be replaced by $\langle i;l\rangle$ pairs in PMCFG for some l. We extend the PGF syntax with a $\langle i;\pi\rangle$ meta-value which is only used internally in the generation. Here, π is a sequence of indices and not only one index as it is the case in the final PMCFG. Like in the operational semantics, the rules define the relation $\gamma \vdash t \Downarrow v$ but this time the context γ is a sequence of assumption sets. Each assumption set a_k contains pairs (π, i) which say that if π is the sequence of indices $l_1 \ldots l_n$ then we assume that $\$_k!l_1 \ldots !l_n \Downarrow i$.

In addition, two sets have to be computed for each category C: $T_s(C)$ and $T_p(C)$. T_s is the set of all sequences of integers $i_1 \ldots i_n$ such that if x is an expression of type C then $x!i_1!i_2!\ldots!i_n$ is of type Str. T_p is the set of all (π,k) pairs where π is again a sequence $i_1 \ldots i_n$ but this time $x!i_1!i_2!\ldots!i_n$ is of type Int_k.

The abstract interpretation rules for strings, integers and tuples are exactly the same but the rule for argument variables is completely new. It says that since we do not know the actual value of the variable we just replace it with the meta-value $\langle k;\ \rangle$. Furthermore there is an additional rule for projection which deals with the case when we have argument variable on the left-hand side of a projection. Basically the rule for argument variables and

Strings:

$$[] \Downarrow [] \quad "foo" \Downarrow "foo" \quad \frac{s \Downarrow v \;\; t \Downarrow w}{s{+}{+}t \Downarrow v{+}{+}w}$$

Bounded integers:

$$i \Downarrow i \; (i = 1,2,\ldots)$$

Tuples:

$$\frac{t_1 \Downarrow v_1 \;\ldots\; t_n \Downarrow v_n}{< t_1,\ldots,t_n > \Downarrow < v_1,\ldots,v_n >}$$

Projections:

$$\frac{t \Downarrow < v_1,\ldots,v_n > \;\; u \Downarrow i}{t!u \Downarrow v_i}\; i = 1,\ldots,n \quad \frac{t \Downarrow \langle k;\pi\rangle \;\; u \Downarrow i}{t!u \Downarrow \langle k;\pi\, i\rangle}\; i = 1,\ldots,n$$

Argument variable:

$$\$_k \Downarrow \langle k;\,\rangle$$

Parameter Substitution:

$$\frac{a_1 \ldots a_n \vdash t \Downarrow \langle k;\pi\rangle}{a_1 \ldots a_k \cup (\pi,i) \ldots a_n, \vdash t \Downarrow i}\; (\pi,m) \in T_p(A_k),\; i = 1,\ldots,m,\; \forall j.((\pi,j) \in a_k \Rightarrow i = j)$$

Table 3 Abstract interpretation of PGF.

the extra projection rule converts terms like $\$_k!l_1 \ldots !l_n$ to meta-values like $\langle k;\pi\rangle$ where π is the sequence $l_1 \ldots l_n$. Since the terms are well-typed at some point either $\pi \in T_s(A_k)$ or $(\pi,n) \in T_p(A_k)$ for some n will be the case. In the first case the meta value will be left unchanged in the evaluated term. The second case is more important because it triggers the parameter substitution rule. This rule is nondeterministic because it makes an arbitrary choice for i and records the choice in the context γ. The last side condition in the rule ensures that if we already had made some assumption for $\langle k;\pi\rangle$ we cannot choose any other value except the value that is already in the context. Since the integers are bounded we have only a finite set of choices which ensures the termination.

Let's use the linearization rule from section 2.4 as an example again:

lin Pred $= \$_1\,!\,1 +\!\!+ (\$_2\,!\,(\$_1\,!\,2))\,!\,(\$_1\,!\,3)$

The first subterm $\$_1\,!\,1$ is simply reduced to $\langle 1;1\rangle$ by the derivation:

$$\frac{\$_1 \Downarrow \langle 1;\rangle \quad 1 \Downarrow 1}{\$_1!1 \Downarrow \langle 1;1\rangle}$$

The derivation of the second subterm is more complex because it contains argument variables on the right hand side of a projection, so they have to be removed using the parameter substitution rule:

$$\frac{\dfrac{a_1\; a_2 \vdash \$_1 \Downarrow \langle 1;\rangle \quad a_1\; a_2 \vdash 2 \Downarrow 2}{a_1\; a_2 \vdash \$_1!2 \Downarrow \langle 1;2\rangle}}{(a_1 \cup \{(2,x)\})\; a_2 \;\vdash\; \$_1!2 \;\Downarrow\; x}$$

Since the parameter substitution is nondeterministic there are two possible derivations but they differ only in the final value, so we use the variable x to denote either value 1 or 2. In a similar way we can deduce that $(a_1 \cup \{(2,x),(3,y)\})\; a_2 \;\vdash\; \$_1!3 \;\Downarrow\; y$, where y is either 1, 2 or 3. Combining the two results the derivation for the second term gives:

$$\frac{\dfrac{\$_2 \Downarrow \langle 2;\rangle \quad \$_1!2 \Downarrow x}{\$_2!(\$_1!2) \Downarrow \langle 2;x\rangle} \quad \$_1!3 \;\Downarrow\; y}{(\$_2!(\$_1!2))!(\$_1!3) \;\Downarrow\; \langle 2;x\,y\rangle}$$

By applying the rule for the concatenation the final result we get:

$$\langle 1;1\rangle \mathbin{++} \langle 2;x\ y\rangle$$

The output from the abstract interpretation can be converted directly to a PMCFG tuple. In well-typed terms a tuple can appear only at the top-level, inside another tuple or on the left-hand side of a record projection. In the abstract interpretation all tuples inside record projections are removed so that the only choice for the evaluated term is to be a tree of nested tuples with leaves of type either Str or Int_n. The tree strictly follows the structure of the linearization type of the result category. The term can be flattened just like we did with the linearization types.

For each possible tree a new unique function is generated with definition containing all leaves of type Str as tuple constituents. For the example above this will lead to six functions:

$$\begin{aligned}
Pred_1 &:= \langle 1;1\rangle\ \langle 2;1\ 1\rangle\\
Pred_2 &:= \langle 1;1\rangle\ \langle 2;2\ 1\rangle\\
Pred_3 &:= \langle 1;1\rangle\ \langle 2;1\ 2\rangle\\
Pred_4 &:= \langle 1;1\rangle\ \langle 2;2\ 2\rangle\\
Pred_5 &:= \langle 1;1\rangle\ \langle 2;1\ 3\rangle\\
Pred_6 &:= \langle 1;1\rangle\ \langle 2;2\ 3\rangle
\end{aligned}$$

The bounded integers in the term are used to determine the right PMCFG result category and each assumption set a_i in the context γ is used to determine the corresponding argument category in the production. One production is generated for every function:

$$\begin{aligned}
S &\rightarrow Pred_1[\mathrm{NP}_{11}, \mathrm{VP}]\\
S &\rightarrow Pred_2[\mathrm{NP}_{21}, \mathrm{VP}]\\
S &\rightarrow Pred_3[\mathrm{NP}_{12}, \mathrm{VP}]\\
S &\rightarrow Pred_4[\mathrm{NP}_{22}, \mathrm{VP}]\\
S &\rightarrow Pred_5[\mathrm{NP}_{13}, \mathrm{VP}]\\
S &\rightarrow Pred_6[\mathrm{NP}_{23}, \mathrm{VP}]
\end{aligned}$$

4.3 Common subexpression elimination in PMCFG

The produced PMCFG could be very big without some form of common subexpression elimination. There are three elimination techniques that are implemented so far and they are described in this section. All of them have been discovered in experiments with real grammars.

The first observation is that the conversion algorithm in the previous section always generates a pair of function definition and production rule using the same function. It happens to be pretty common that one function could be reused in two different productions because the original definitions are equivalent. In the real implementation first the definition is generated and after that it is compared with the already existing definitions. Only if it is distinct a new function is generated.

Another issue is that there are many constituents in the function bodies which are equal but are used in different places. For that reason we collect a list of distinct constituents and

Language	Productions			File Size (Kb)		
	Plain	Optimized	Ratio	Plain	Optimized	Ratio
Bulgarian	3629	3516	1.03	20359	5021	4.05
Danish	1696	1615	1.05	1399	593	2.36
English	1198	1165	1.03	1648	710	2.32
Finnish	-	141441	-	-	6357	-
German	11079	8078	1.37	56559	3027	18.68
Italian	-	1089621	-	-	106282	-
Norwegian	1773	1696	1.05	1418	596	2.38
Russian	5248	5077	1.03	7735	1261	6.13
Swedish	1535	1496	1.03	1161	577	2.01

Table 4 Grammar sizes in number of PMCFG productions and PGF file size, for the GF Resource Grammar Library.

then the function definitions are rewritten to contain only the indices of the corresponding constituents.

The last observation is that there are groups of productions like:

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

where the list could be very large. This happens when in some linearization function some parameters are used only when another parameter has a specific value. The abstract interpretation could not detect all these cases. It detects only the parameters that are not used at all and then the conversion rules are introduced. The list of productions can be compressed by introducing extra conversion rules:

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

These optimizations have been implemented and tried with the resource grammar library (Ranta 2008), which is the largest collection of grammars written in GF. The produced grammar sizes are summarized in Table 4. The first column shows the grammar size in number of PMCFG productions and the second the total PGF file size.

The common subexpression optimization seems to reduce the file size from 2 to 18 times depending on the grammar. For two of the languages Finnish and Italian it is even impossible to compile the grammar without the optimization. The conversion was tried on a computer with 2 GB physical memory but it was not enough to fit the unoptimized grammar. The conversion for Italian is possible but currently it takes 2 days on a 2 GHz CPU. For that reason three other Romance languages are not listed: Catalan, French and Spanish. Their compilation should be possible but would probably require the same amount of time as Italian. The main problem is the compilation of the SlashVP rule. All Romance languages have clitic structures in the verb phrases which cause exponential growth of the grammar size. The same applies to Interlingua which is an artificial language whose verb phrases also have clitics.

Some statistics were collected from the compiled Italian grammar which suggest directions for further optimizations. The production and function definitions generated from the SlashVP linearization function constitute 41% of the total grammar size. In the SlashVP size the dominant proportion is due to the PMCFG functions. There are 11280 functions where each has 321 constituents. Fortunately only 5% of these constituents are distinct. This suggests that the data is sparse and there should be a more compact representation for it. This also explains why the compilation is so slow. Each constituent is currently compiled independently and this means a lot of repeated work. If some kind of memoization were used, the compilation time could be reduced dramatically.

4.4 Parsing with PMCFG

Efficient recognition and parsing algorithms for MCFG have been described in (Nakanishi et al. 1997), Ljunglöf (2004) and (Burden and Ljunglöf 2005). MCFG is a linear form of PMCFG where each constituent of an argument is used exactly once. Ljunglöf (2004) gives an algorithm for approximating a PMCFG with an MCFG. With the approximation it is possible to use a parsing algorithm for MCFG to parse with a PMCFG, but after that a postprocessing step is needed to recover the original parse tree and to filter out spurious parse trees via unification. Instead we are using a parsing algorithm that works directly with PMCFG and also has the advantage that it is incremental. The incrementality is important because it can be used for word prediction (see section 5.1). The incremental algorithm itself will appear in a separate paper.

4.5 Parse trees

The output from the parser is a syntax tree or a set of trees where the nodes are labeled with PMCFG functions. The trees have to be converted back to PGF abstract expression. In the absence of high-order terms (section 3.3) the transformation is trivial. The definition of each PMCFG function is annotated with the corresponding PGF linearization function so they just have to be replaced. The PMCFG grammar might be erasing i.e. some argument might not be used at all. In this case the slot for this argument is filled with meta variable.

5 Using PGF

In this section, we will outline how PGF grammars can be used to construct natural language processing applications. We first list a number of operations that can be performed with a PGF grammar, along with some possible applications of these operations. We then give a brief overview of the APIs (Application Programmer's Interfaces) which are currently available for using PGF functionality in applications. Finally, we outline how PGF grammars can be produced from the more grammar-writer friendly format GF.

5.1 PGF operations

PGF grammars can be used for a wide range of tasks, either stand-alone, or as integral parts of a larger application. Figure 2 shows an overview of how PGF grammars can be used in natural language processing applications.

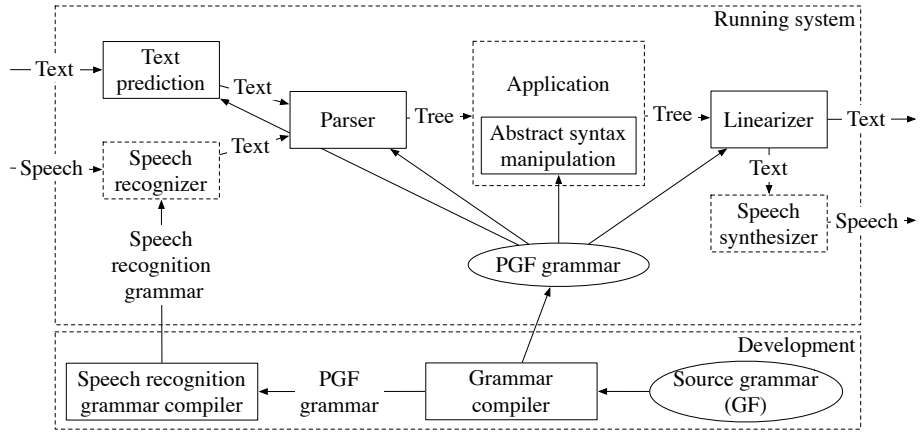

Fig. 2 Overview of PGF applications.

Parsing takes a string in some language and produces zero (in the case of an out-of-grammar input), one (in the the case of an unambiguous input), or more (for ambiguous inputs) abstract syntax trees. PGF parsing is for example useful for handling natural language user input in applications. This lets the rest of the application work on abstract syntax trees, which makes it easy to localize the application to new languages.

Text Prediction is related to parsing. The incremental PMCFG parsing algorithm can parse an incomplete input, and return the set of possible next tokens. If the last token in the given input is itself incomplete, the list of complete tokens can be given to the parser, and the result filtered to retain only those tokens that have the last (incomplete) token as a prefix.

Linearization, the production of text given an abstract syntax tree, can be used to produce natural language system outputs, either as text, or, by using speech synthesis software, speech output. In the latter case, the concrete syntax may contain annotations such as SSML tags which help the speech synthesizer.

Translation is the combination of parsing and linearization (Khegai 2006).

Abstract syntax generation, is made easy by the PGF abstract syntax type system. A potentially infinite lazy list of abstract syntax trees can be generated randomly, or exhaustively through iterative deepening. Random generation can be used to generate a monolingual corpus (generate a list of random abstract syntax trees and linearize them to given language), a multilingual parallel corpus (generate trees and linearize to several languages), or a treebank (generate trees, linearize, and output text, tree pairs).

Abstract syntax manipulation is useful in any application that analyzes or produces abstract syntax trees. The PGF type system makes it possible to expose abstract syntax manipulation to the user in a safe way. When statically typed languages such as Haskell or Java are used, it is possible to generate host language data types from an Figure 3 shows such data types generated from the example abstract syntax in Section 2.2.

5.2 PGF Interpreter API

A PGF Interpreter API allows an application programmer to make use of PGF grammars for the tasks listed above in a program written in some general purpose programming language. There are currently APIs for Haskell, Java, JavaScript, Prolog, C, C++, and web applications,

```
data S  = Pred NP VP
data NP = John
data VP = Walk
```

```
abstract class S {...}
class Pred extends S { NP np; VP vp; ...}
abstract class NP {...}
class John extends NP {...}
abstract class VP {...}
class Walk extends VP {...}
```

Fig. 3 Haskell and Java data types for the abstract syntax in Section 2.2.

```
readPGF :: FilePath → IO PGF
linearize :: PGF → Language → Tree → String
parse :: PGF → Language → Category → String → [Tree]
generateRandom :: PGF → Category → IO [Tree]
generateAll :: PGF → Category → [Tree]
complete :: PGF → Language → Category → String → [String]
```

Fig. 4 A part of the Haskell PGF Interpreter API.

```
public class PGF {
  public static PGF readPGF (String path)
  public String linearize (String lang, Tree tree)
  public List<Tree> parse (String lang, String text)
}
```

Fig. 5 A part of the Java PGF Interpreter API.

with varying degrees of functionality. We examine the Haskell API in some detail. The other APIs, some of which are covered briefly below, have the same functionality, or subsets of it.

Haskell API Figure 4 shows the most important functions in the Haskell PGF Interpreter API. There are also functions for manipulating Tree value, for getting metadata about grammars, and variants of generation functions which give more control over the generation process, for example by setting a maximum tree height.

Java PGF Interpreter API The Java API is very similar to the Haskell API, but with a more object-oriented design, see Figure 5.

JavaScript PGF Interpreter API PGF grammars can also be converted into a JavaScript format, for which there is an interpreter that implements linearization, parsing and abstract syntax tree manipulation (Meza Moreno 2008).

PGF Interpreter Web Service PGF parsing, linearization and translation is also available as a web service, in the form of a FastCGI (Brown 1996) program with a RESTful (Fielding 2000) interface that returns JSON (Crockford 2006) structures. For example, a request to translate the sentence *this fish is fresh* from the concrete syntax `FoodEng` to all available concrete syntaxes may return the following JSON structure:

```
[{"from":"FoodEng","to":"FoodEng","text":"this fish is fresh"},
 {"from":"FoodEng","to":"FoodIta","text":"questo pesce è fresco"}]
```

5.3 Compiling PGF to other formats

The declarative nature of PGF makes it possible to translate PGF grammars to other grammar formalisms. It is theoretically interesting to produce algorithms for translating between different grammar formalisms. However, it also has practical applications, as it lets us use PGF grammars with existing software systems based on other grammar formalisms, for example speech recognizers (Bringert 2007a). When combined with PGF parsing, this makes it possible for an application to accept speech input, based solely on the information in the PGF grammar. Examples of such applications include dialogue systems (Ericsson et al. 2006; Bringert 2007b) and speech translation (Bringert 2008).

Most examples in the rest of this section will be based on the PGF grammar shown in Figure 7, which extends the earlier example grammar with the And and We functions. This grammar has been chosen to compactly include both agreement and left-recursion (at the expense of ambiguous parsing, this could be fixed with a slightly larger grammar).

5.3.1 Context-free approximation

A context-free grammar (CFG) that approximates a PGF grammar can be produced by first producing a PMCFG as described earlier. The PMCFG is then approximated with a CFG by converting each PMCFG category-field pair to a CFG category. In the general case, this is an approximation, since PMCFG is a stronger formalism than CFG. The example language with agreement is converted to the CFG shown in Figure 8.

The PMCFG grammar given in Section 4.1 for the context-sensitive language $a^n b^n c^n$ is approximated by the context-free grammar shown in Figure 14. In this case the context-free approximation is overgenerating, as it generates the language $a^* b^* c^*$. The simpler $a^n b^n$ language is context-free, but may be described by either a context-sensitive grammar in a similar way to the $a^n b^n c^n$ language above, or by a context-free grammar. The context-free approximation will preserve the language described by an already context-free grammar, but not necessarily the language of a context-sensitive grammar that defines a context-free language. It is not possible to devise an algorithm that converts all context-sensitive grammars that generate context-free languages to context-free grammars, since deciding whether a context-sensitive language is context-free is undecidable. We conjecture that this also holds for deciding whether a PMCFG generates a context-free language.

5.3.2 Context-free transformations

Our PGF compiler also implements a number of transformations on context-free grammars, such as cycle elimination, bottom-up and top-down filtering, left-recursion elimination, identical category elimination, and EBNF compaction. This makes it possible to produce grammars in a number of restricted context-free formats as required by speech recognition software.

```
cat S; NP; VP
fun And      : S → S → S
    Pred     : NP → VP → S
    John, We : NP
    Walk     : VP
param Num = Sg | Pl
param Pers = P1 | P2 | P3
lincat S    = Str
       NP   = {s : Str; n : Num; p : Pers}
       VP   = Num ⇒ Pers ⇒ Str
lin And x y = x ++ "und" ++ y
    Pred np vp = np.s ++ vp ! np.n ! np.p
    John = {s = "John"; n = Sg; p = P3}
    We   = {s = "wir"; n = Pl; p = P1}
    Walk =
      table { Sg ⇒ table {P1 ⇒ "gehe";
                          P2 ⇒ "gehst";
                          P3 ⇒ "geht"};
              Pl ⇒ table {P1 ⇒ "gehen";
                          P2 ⇒ "geht";
                          P3 ⇒ "gehen"}}
```

Fig. 6 GF grammar.

```
cat S; NP; VP
fun And  : (S, S) → S
    Pred : (NP, VP) → S
    John : () → NP
    We   : () → NP
    Walk : () → VP
lincat S  = Str
       NP = Str * Int2 * Int3
       VP = (Str * Str * Str)
            * (Str * Str * Str)
lin And  = $1 ++ "und" ++ $2
    Pred = $1 ! 1 ++ ($2 ! ($1 ! 2)) ! ($1 ! 3)
    John = <"John", 1, 3>
    We   = <"wir", 2, 1>
    Walk =
      < < "gehe",
          "gehst",
          "geht">,
        <"gehen",
          "geht",
          "gehen" > >
```

Fig. 7 PGF grammar.

```
S    → S "und" S | NP1 VP2 | NP2 VP1
NP1 → "John"
NP2 → "wir"
VP1 → "gehen"
VP2 → "geht"
```

Fig. 8 CFG.

```
S    → S "und" S { And ($1, $2) }
     | NP1 VP2 { Pred ($1, $2) }
     | NP2 VP1 { Pred ($1, $2) }
NP1 → "John" { John () }
NP2 → "wir" { We () }
VP1 → "gehen" { Walk () }
VP2 → "geht" { Walk () }
```

Fig. 9 Fig. 8 with interpretation.

```
S    → NP1 S2 | NP2 S3
S2   → VP2 | VP2 S4
S3   → VP1 | VP1 S4
S4   → "und" S | "und" S S4
NP1 → "John"
NP2 → "wir"
VP1 → "gehen"
VP2 → "geht"
```

Fig. 10 Non-left-recursive CFG.

```
S    → NP1 S2 { $2 ($1) }
     | NP2 S3 { $2 ($1) }
S2   → VP2 { λx.Pred (x, $1) }
     | VP2 S4 { λx.$2 (Pred (x, $1)) }
S3   → VP1 { λx.Pred (x, $1) }
     | VP1 S4 { λx.$2 (Pred (x, $1)) }
S4   → "und" S { λx.And (x, $1) }
     | "und" S S4 { λx.$2 (And (x, $1)) }
NP1 → "John" { John () }
NP2 → "wir" { We () }
VP1 → "gehen" { Walk () }
VP2 → "geht" { Walk () }
```

Fig. 11 Fig. 10 with interpretation.

```
S  → NP1 VP2 S2 | NP2 VP1 S2
S2 → "und" S | ε
NP1 → "John"
NP2 → "wir"
VP1 → "gehen"
VP2 → "geht"
```

Fig. 12 Regular grammar.

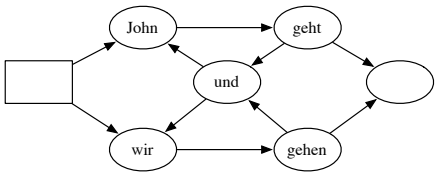

Fig. 13 Finite-state automaton.

$S \rightarrow A_0\ A_1\ A_2$
$A \rightarrow A_0 \mid A_1 \mid A_2$
$A_0 \rightarrow \varepsilon \mid$ "a" A_0
$A_1 \rightarrow \varepsilon \mid$ "b" A_1
$A_2 \rightarrow \varepsilon \mid$ "c" A_2

Fig. 14 Context-free approximation of the PMCFG grammar for the $a^n b^n c^n$ language.

5.3.3 Embedded tree building code

The rules for producing abstract syntax trees can be preserved through the grammar translations listed above. This makes it possible to include abstract syntax tree building code in the generated context-free grammars, for example in SISR (Burke and Van Tichelen 2006) format. Figure 9 shows such an annotated grammar. As is shown in Figure 11, the abstract syntax tree building annotations can be carried through the left-recursion removal transformation, by the use of λ-terms.

5.3.4 Regular / finite-state approximation

The PGF grammar compiler can also approximate the produced context-free grammars with regular grammars, which can in turn be compiled to finite-state automata. An example of this is shown in Figures 12 and 13. This lets us support speech recognizers which require regular or finite-state language models, or for typical finite-state natural language processing tasks such as marking noun phrases in a corpus.

5.4 Compiling GF to PGF

While PGF is suitable for efficient and simple implementation, PGF grammars are not meant to be produced by humans. Rather, it is intended as the target language of a compiler from a high-level grammar language. In this section, we will outline the differences between the human-friendly GF language, and the machine-friendly PGF language, and how grammars written in GF can be translated to PGF. The PGF syntax and semantics can be seen as a subset of the GF syntax and semantics, while the PGF type system is less strict than that of GF. Where the GF type system is concerned with correctness, the PGF type system only ensures safety.

5.4.1 Tables and records

In GF, there are two separate constructs that correspond to PGF tuples: tables and records. In GF, tables are tuples whose values all have the same type, and finite algebraic data types or records are used to select values from tables. Tables are used to implemented parametric features such as inflection.

Records, on the other hand, can have values of different type, but the selectors used with records are labels which must be known statically. Records are used to implement inherent features and discontinuous constituents.

Both tables and records can be nested, but they always have a statically known size, which only depends on their type. This makes it possible to translate both records and tables

to PGF tuples. For example, the GF grammar shown in Figure 6 is translated to the PGF grammar in Figure 7.

5.4.2 Structured parameter values and labels

As noted above, in GF, table projection is done with structured values known as parameters. These can be combinations of non-recursive algebraic data types and records of parameters. Since parameter records contain a known number of values, and the algebraic parameter values are non-recursive, each parameter type contains a finite number of values. This makes it possible to translate each parameter type to a bounded integer type, suitable for projection on PGF tuples.

5.4.3 Stricter type checking

As noted above, the GF concrete syntax type system is stricter than the PGF type system. In PGF, bounded integers are used to represent all parameter types and record labels, which means that many distinctions made by the GF type checker are not available in PGF. The differences between the GF and PGF type system can be compared to the differences between the Java type system and JVM bytecode verification. The type system for abstract syntax is identical in GF and PGF.

5.4.4 Pattern matching

Table projection in GF can be done by pattern matching with rather complex patterns. When compiling to PGF, all tables are expanded to have one branch with for each possible parameter value, to allow for translation to PGF tuples.

5.4.5 Modularity

GF grammars are organized into modules which can be compiled to core GF separately, like Java classes or C object files. PGF on the other hand has no module system and is a single file, similar to a statically linked executable. This simplifies PGF implementations and makes it easy to distribute PGF grammars.

5.4.6 Auxiliary operations

In GF, auxiliary operations can be defined in order to implement for example morphological paradigms or common syntactic operations. These operations, like all GF concrete syntax can include complex features such as higher-order functions and pattern matching on strings. During the translation to PGF, all auxiliary operations are inlined, and all expressions are evaluated to produce valid PGF terms. This means, for example, that all strings which are used in pattern matching or in-token concatenation must be statically computable.

Because this inlining and computation can produce very large and repetitive PGF terms, common sub-expression elimination is performed. This can recover some of the auxiliary operations, but in a simpler form, but it can also discover new opportunities for code sharing, as PGF macros allow open terms, are type agnostic, and shared between code that comes from unrelated GF modules.

6 Results and evaluation

6.1 Systems using PGF

A number of natural language applications have been implemented using PGF or its predecessor GFCM. Above, we have listed a number of individual tasks that can be performed with PGF grammars. However, realistic applications often make use PGF for more than one task. This helps avoid the duplicated work involved in manually implementing multiple components which cover the same language fragment.

GOTTIS (Bringert et al. 2005) is an experimental multimodal and multilingual dialogue system for public transportation queries. It uses the a generated Nuance GSL speech recognition grammar for speech input, embedded parsing and linearization for system input and output, and generated Java data types for analysing input abstract syntax trees and producing output abstract syntax trees. Both input and output in GOTTIS make use of *multimodal* grammars. The input grammar allows *integrated multimodality*, e.g. "I want to go here", accompanied by a click on a map. This is implemented by using a two-field record (PGF tuple) to represent spatial expressions, where one field contains the spoken component, and another contains the click component. System output makes use of *parallel multimodality*; one concrete syntax produces spoken route descriptions, and another produces drawing instructions which are used to display routes on a map. Other examples of using GF grammars for formal languages include the WebALT mathematics problem editor (Cohen et al. 2006) and the KeY software specification editor (Burke and Johannisson 2005),

PGF abstract syntax can be used as a specification of a dialogue manager (Ranta and Cooper 2004). Together with the existing speech recognition grammar generation with embedded semantic interpretation tags, and the PGF to JavaScript compiler, this can be used to generate complete multimodal and multilingual VoiceXML dialogue systems (Bringert 2007b).

DUDE (Lemon and Liu 2006) and its extension REALL-DUDE (Lemon et al. 2006) are environments where non-experts can develop dialogue systems based on Business Process Models describing the applications. From keywords, prompts and answer sets defined by the developer, the system generates a GF grammar. This grammar is used for parsing input, and for generating a language model in SLF or GSL format.

Several syntax-directed editors for controlled languages (Khegai et al. 2003; Johannisson et al. 2003; Meza Moreno and Bringert 2008), have been implemented using PGF and its predecessors. They make use of abstract syntax manipulation, parsing linearization.

PGF can be used to implement complete limited domain speech translation systems that use PGF to produce speech recognition grammars and to perform parsing and linearization (Bringert 2008).

Text prediction can be used to implement text-based user interfaces which can show the user what inputs are allowed by the grammar. Examples of applications where this might be useful are editors for controlled languages, language learning software or limited domain translation software such as tourist phrase books. A web-based controlled language translator using text prediction is currently being developed.

Jonson (2006, 2007) used random corpus generation to produce statistical language models (SLM) for speech recognition from a domain-specific grammar. When combined with a general SLM trained on existing general domain text, the precision on in-grammar inputs was largely unchanged, while out-of-grammar inputs where handled with much higher precision, compared to a pure grammar-based language model. When producing corpora for training SLMs, dependently typed abstract syntax can be used to reduce over-generation (Jon-

son 2006). Other, as yet unexplored, applications of PGF abstract syntax generation are the use of generated multilingual parallel corpora for training statistical machine translation systems, and the use of generated treebanks for training statistical parsers.

The grammars from the GF resource grammar library (Ranta 2008) can be used not only to implement application-specific grammars, but also as general wide-coverage grammars. For example, the English resource grammar, with a large lexicon and some minor syntax extensions, covers a significant portion of the sentences in the FraCaS semantic test suite (Cooper et al. 1996).

7 Related work

Compilation of grammars to low-level formats is an old idea. The most well-known examples are parser generators in the YACC family (Johnson 1975). The output of YACC-like parser generation is a piece of host language source code, which can be seamlessly integrated in a program written in the host language. YACC-like systems for full context-free grammars suitable for natural language include the work by Tomita and Carbonell (1987), the NLYACC system (Ishii et al. 1994), and the GLR extension of the Happy parser tool for Haskell (Callaghan and Medlock 2004). The main differences between PGF and the YACC family are that PGF is stronger than context-free grammars, that PGF contains no host language code and is therefore portable to many host languages, that PGF supports linearization in addition to parsing, and that PGF grammars can be multilingual.

HPSG (Pollard and Sag 1994) and LFG (Bresnan 1982) are grammar formalisms used for large-scale grammar writing and processing. In their implementation, the use of optimizing compilers is essential, to support at the same time high-level grammar writing and efficient run-time execution. HPSG compilers, for instance, have used advanced compiler techniques such as data flow analysis (Minnen et al. 1995). The main focus in both grammars is parsing, but also generation is supported. Both in HPSG and LFG, systems of parallel grammars for different languages have been developed, but neither formalism is multilingual in the way GF/PGF is. The currently most popular implementations are LKB (Copestake 2002) for HPSG and XLE (Kaplan and Maxwell 2007) for LFG. To our knowledge, neither formalism supports generation of portable low-level code.

An emerging species of embedded grammar applications is language models for speech recognition. Regulus (Rayner et al. 2006) is a system that compiles high-level feature-based grammars into low-level context-free grammars in the Nuance format (Nua 2003). PGF can likewise be compiled into Nuance and a host of other speech grammar formats (Bringert 2007a).

Type-theoretical grammar formats based on Curry's distinction between tectogrammar and phenogrammar have gained popularity in the recent years: ACG (Abstract Categorial Grammars) (de Groote 2001), HOG (Higher-Order Grammars) (Pollard 2004), and Lambda Grammars (Muskens 2001) are the most well-known examples besides GF. The work in implementing these formalisms has not come so far as in GF. One obvious idea for implementing them is to use compilation to PGF. But it remains to be seen if PGF is general enough to support this. For instance, ACG is more general than PGF in the sense that linearization types can be function types, but less general in the sense that functions have to be linear (that is, use every argument exactly once). This means that the style of defining functions is very different, for instance, that a rule written with multiple occurrences of a variable in PGF is in ACG encoded as a higher-order function.

PMCFG, while known for almost two decades and having nice computational properties, has not been used for practical grammar writing. Even the use of PMCFG as a target format for grammar compilation seems to be new to the GF project.

8 Conclusion

PGF was first created as a low-level target format for compiling high-level grammars written in GF. The division between high-level source formats and low-level target formats has known advantages in programming language design, which have been confirmed in the case of GF and PGF. One distinguishing property is redundancy: the absence of redundancy from PGF makes is maximally easy to write PGF interpreters, to compile PGF to other formats, and to reason about PGF. In GF, on the other hand, computationally redundant features, such as intensional type distinctions, inlinable functions, and separately compilable modules, support the programmer's work by permitting useful error messages and an abstract programming style. GF as compiled to PGF has made it possible to build grammar-based systems that combine linguistic coverage (the GF resource grammar library) with efficient run-time behaviour (mostly linear-time generation, incremental PMCFG parsing) and integration with other system components (web pages via JavaScript, spoken language models via context-free approximations). For other grammar formalisms than GF, compilation to PGF could be used as an economical implementation technique, which would moreover make it possible to link together grammars written in different high-level formalisms.

References

Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, December 1997. ISBN 0521582741.

J. Bresnan, editor. *The Mental Representation of Grammatical Relations*. MIT Press, 1982.

Björn Bringert. Speech Recognition Grammar Compilation in Grammatical Framework. In *Proceedings of the Workshop on Grammar-Based Approaches to Spoken Language Processing, Prague, Czech Republic*, pages 1–8. Association for Computational Linguistics, June 2007a. URL http://www.aclweb.org/anthology/W/W07/W07-1801.

Björn Bringert. Rapid Development of Dialogue Systems by Grammar Compilation. In Simon Keizer, Harry Bunt, and Tim Paek, editors, *Proceedings of the 8th SIGdial Workshop on Discourse and Dialogue, Antwerp, Belgium*, pages 223–226. Association for Computational Linguistics, September 2007b. URL http://www.sigdial.org/workshops/workshop8/Proceedings/SIGdial39.pdf.

Björn Bringert. Speech Translation with Grammatical Framework. In *Coling 2008: Proceedings of the workshop on Speech Processing for Safety Critical Translation and Pervasive Applications, Manchester, UK*, pages 5–8. Coling 2008 Organizing Committee, August 2008. URL http://www.cs.chalmers.se/~bringert/publ/gf-slt/gf-slt.pdf.

Björn Bringert, Robin Cooper, Peter Ljunglöf, and Aarne Ranta. Multimodal Dialogue System Grammars. In *Proceedings of DIALOR'05, Ninth Workshop on the Semantics and Pragmatics of Dialogue, Nancy, France*, pages 53–60, June 2005. URL http://dialor05.loria.fr/Papers/07-BjornBringert.pdf.

Mark R. Brown. FastCGI: A High-Performance Gateway Interface. In Anton Eliëns, editor, *Programming the Web - a search for APIs, Fifth International World Wide Web Conference (WWW5), Paris, France*, May 1996. URL http://www.cs.vu.nl/~eliens/WWW5/papers/FastCGI.html.

Håkan Burden and Peter Ljunglöf. Parsing Linear Context-Free Rewriting Systems. In *Proceedings of the Ninth International Workshop on Parsing Technology*, pages 11–17, Vancouver, British Columbia, 2005. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/W/W05/W05-1502.

David Burke and Luc Van Tichelen. Semantic Interpretation for Speech Recognition (SISR) Version 1.0. Working draft, W3C, November 2006. URL http://www.w3.org/TR/2006/WD-semantic-interpretation-20061103.

David A. Burke and Kristofer Johannisson. Translating Formal Software Specifications to Natural Language. In *Logical Aspects of Computational Linguistics*, volume 3492 of *Lecture Notes in Computer Science*, pages 51–66. Springer, Heidelberg, May 2005. doi: 10.1007/11422532_4. URL http://dx.doi.org/10.1007/11422532_4.

P. Callaghan and B. Medlock. Happy-GLR, 2004. URL http://www.dur.ac.uk/p.c.callaghan/happy-glr/.

Arjeh Cohen, Hans Cuypers, Karin Poels, Mark Spanbroek, and Rikko Verrijzer. WExEd - WebALT Exercise Editor for Multilingual Mathematical Exercises. In Mika Seppälä, Sebastian Xambo, and Olga Caprotti, editors, *WebALT 2006, First WebALT Conference and Exhibition, Eindhoven, The Netherlands*, pages 141–145, January 2006. URL http://www.win.tue.nl/~amc/pub/wexed.pdf.

Robin Cooper, Dick Crouch, Jan van Eijck, Chris Fox, Josef van Genabith, Jan Jaspars, Hans Kamp, David Milward, Manfred Pinkal, Massimo Poesio, Steve Pulman, Ted Briscoe, Holger Maier, and Karsten Konrad. A Semantic Test Suite. In *Using the Framework, Deliverable D16*, chapter 3. The FRACAS Consortium, January 1996. URL ftp://ftp.cogsci.ed.ac.uk/pub/FRACAS/del16.ps.gz.

A. Copestake. *Implementing Typed Feature Structure Grammars*. CSLI Publications, 2002.

Douglas Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July 2006. URL http://www.ietf.org/rfc/rfc4627.txt.

Haskell B. Curry. Some Logical Aspects of Grammatical Structure. In Roman O. Jakobson, editor, *Structure of Language and its Mathematical Aspects, volume 12 of Symposia on Applied Mathematics*, pages 56–68. American Mathematical Society, Providence, 1961.

Philippe de Groote. Towards abstract categorial grammars. In *Proceedings of 39th Annual Meeting of the Association for Computational Linguistics, Toulouse, France*, pages 252–259, Morristown, NJ, USA, July 2001. Association for Computational Linguistics. doi: 10.3115/1073012.1073045. URL http://dx.doi.org/10.3115/1073012.1073045.

Stina Ericsson, Gabriel Amores, Björn Bringert, Håkan Burden, Ann-Charlotte Forslund, David Hjelm, Rebecca Jonson, Staffan Larsson, Peter Ljunglöf, Pilar Manchón, David Milward, Guillermo Pérez, and Mikael Sandin. Software illustrating a unified approach to multimodality and multilinguality in the in-home domain. deliverable 1.6, 2006. URL http://www.talk-project.org/fileadmin/talk/publications_public/deliverables_public/D1_6.pdf.

Roy T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. URL http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *JACM*, 40(1):143–184, 1993.

Masayuki Ishii, Kazuhisa Ohta, and Hiroaki Saito. An efficient parser generator for natural language. In *Proceedings of the 15th conference on Computational linguistics*, pages 417–420, Morristown, NJ, USA, 1994. Association for Computational Linguistics. doi: 10.3115/991886.991959.

Kristofer Johannisson, Janna Khegai, Markus Forsberg, and Aarne Ranta. From Grammars to Gramlets. In *The Joint Winter Meeting of Computing Science and Computer Engineering*. Chalmers University of Technology, 2003.

S. C. Johnson. Yacc — yet another compiler compiler. Technical Report CSTR-32, AT & T Bell Laboratories, Murray Hill, NJ, 1975.

Rebecca Jonson. Generating Statistical Language Models from Interpretation Grammars in Dialogue Systems. In *EACL 2006, 11st Conference of the European Chapter of the Association for Computational Linguistics*, 2006. URL http://acl.ldc.upenn.edu/E/E06/E06-1008.pdf.

Rebecca Jonson. Grammar-based context-specific statistical language modelling. In *Proceedings of the Workshop on Grammar-Based Approaches to Spoken Language Processing*, pages 25–32, Prague, Czech Republic, June 2007. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/W/W07/W07-1804.

A. Joshi, K. Vijay-Shanker, and D. Weir. The convergence of mildly context-sensitive grammar formalisms. In P. Sells, S. Shieber, and T. Wasow, editors, *Foundational Issues in Natural Language Processing*, pages 31–81. MIT Press, 1991.

R. Kaplan and J. Maxwell. XLE Project Homepage, 2007. URL http://www2.parc.com/isl/groups/nltt/xle/.

Janna Khegai. Grammatical Framework (GF) for MT in sublanguage domains. In *Proceedings of EAMT-2006, 11th Annual conference of the European Association for Machine Translation, Oslo, Norway*, pages 95–104, June 2006. URL http://www.mt-archive.info/EAMT-2006-Khegai.pdf.

Janna Khegai, Bengt Nordström, and Aarne Ranta. Multilingual Syntax Editing in GF. In Alexander Gelbukh, editor, *Computational Linguistics and Intelligent Text Processing*, volume 2588 of *Lecture Notes in Computer Science*, pages 199–204. 2003. doi: 10.1007/3-540-36456-0_48. URL http://dx.doi.org/10.1007/3-540-36456-0_48.

Oliver Lemon and Xingkun Liu. DUDE: a Dialogue and Understanding Development Environment, mapping Business Process Models to Information State Update dialogue systems. In *EACL 2006, 11st Conference of the European Chapter of the Association for Computational Linguistics*, 2006. URL http://www.aclweb.org/anthology-new/E/E06/E06-2004.pdf.

Oliver Lemon, Xingkun Liu, Daniel Shapiro, and Carl Tollander. Hierarchical Reinforcement Learning of Dialogue Policies in a development environment for dialogue systems: REALL-DUDE. In *BRANDIAL'06, Proceedings of the 10th Workshop on the Semantics and Pragmatics of Dialogue*, pages 185–186, September 2006. URL http://www.ling.uni-potsdam.de/brandial/Proceedings/brandial06_lemon_etal.pdf.

Peter Ljunglöf. *Expressivity and Complexity of the Grammatical Framework*. PhD thesis, Göteborg University, Göteborg, Sweden, 2004. URL http://www.ling.gu.se/~peb/pubs/p04-PhD-thesis.pdf.

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

J. McCarthy. Towards a mathematical science of computation. In *Proceedings of the Information Processing Cong. 62*, pages 21–28, Munich, West Germany, August 1962. North-Holland.

Moisés S. Meza Moreno. Implementation of a JavaScript Syntax Editor and Parser for Grammatical Framework. Master's thesis, Chalmers University of Technology, 2008.

Moisés S. Meza Moreno and Björn Bringert. Interactive Multilingual Web Applications with Grammatical Framework. In Bengt Nordström and Aarne Ranta, editors, *Advances in Natural Language Processing, 6th International Conference, GoTAL 2008, Gothenburg, Sweden*, volume 5221 of *Lecture Notes in Computer Science*, pages 336–347, Heidelberg, August 2008. Springer. doi: 10.1007/978-3-540-85287-2_32. URL http://dx.doi.org/10.1007/978-3-540-85287-2_32.

G. Minnen, D. Gerdemann, and T. Gotz. Off-line optimization for earleystyle hpsg processing, 1995. URL citeseer.ist.psu.edu/article/minnen95offline.html.

R. Montague. *Formal Philosophy*. Yale University Press, New Haven, 1974. Collected papers edited by Richmond Thomason.

R. Muskens. Lambda Grammars and the Syntax-Semantics Interface. In R. van Rooy and M. Stokhof, editors, *Proceedings of the Thirteenth Amsterdam Colloquium*, pages 150–155, Amsterdam, 2001.

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.

Nuance Speech Recognition System 8.5: Grammar Developer's Guide. Nuance Communications, Inc., Menlo Park, CA, USA, December 2003.

C. Pollard and I. Sag. *Head-Driven Phrase Structure Grammar*. University of Chicago Press, 1994.

Carl Pollard. Higher-Order Categorial Grammar. In *Proceedings of Categorial Grammars 2004*, pages 340–361, June 2004. URL http://www.ling.ohio-state.edu/~hana/hog/pollard2004-CG.pdf.

A. Ranta. GF Resource Grammar Library, 2008. URL http://digitalgrammars.com/gf/lib/.

A. Ranta. *Type Theoretical Grammar*. Oxford University Press, 1994.

Aarne Ranta. Grammatical Framework: A Type-Theoretical Grammar Formalism. *Journal of Functional Programming*, 14(2):145–189, March 2004. ISSN 0956-7968. doi: 10.1017/S0956796803004738. URL http://dx.doi.org/10.1017/S0956796803004738.

Aarne Ranta and Robin Cooper. Dialogue Systems as Proof Editors. *Journal of Logic, Language and Information*, 13(2):225–240, 2004. ISSN 0925-8531. doi: 10.1023/B:JLLI.0000024736.34644.48. URL http://dx.doi.org/10.1023/B:JLLI.0000024736.34644.48.

Manny Rayner, Beth A. Hockey, and Pierrette Bouillon. *Putting Linguistics into Speech Recognition: The Regulus Grammar Compiler*. CSLI Publications, Ventura Hall, Stanford University, Stanford, CA 94305, USA, July 2006. ISBN 1575865262.

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. doi: 10.1093/ietisy/e91-d.2.209. 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. doi: 10.1016/0304-3975(91)90374-B. URL http://dx.doi.org/10.1016/0304-3975(91)90374-B.

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. doi: 10.3115/981574.981592. URL http://acl.ldc.upenn.edu/P/P93/P93-1018.pdf.

Masaru Tomita and Jaime G. Carbonell. The universal parser architecture for knowledge-based machine translation. In *IJCAI*, pages 718–721, 1987.