Grammar Development in GF

Aarne Ranta and **Krasimir Angelov** and **Björn Bringert**[*]
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
{aarne,krasimir,bringert}@chalmers.se

Abstract

GF is a grammar formalism that has a powerful type system and module system, permitting a high level of abstraction and division of labour in grammar writing. GF is suited both for expert linguists, who appreciate its capacity of generalizations and conciseness, and for beginners, who benefit from its static type checker and, in particular, the GF Resource Grammar Library, which currently covers 12 languages. GF has a notion of multilingual grammars, enabling code sharing, linguistic generalizations, rapid development of translation systems, and painless porting of applications to new languages.

1 Introduction

Grammar implementation for natural languages is a challenge for both linguistics and engineering. The linguistic challenge is to master the complexities of languages so that all details are taken into account and work seamlessly together; if possible, the description should be concise and elegant, and capture the linguist's generalizations on the level of code. The engineering challenge is to make the grammar scalable, reusable, and maintainable. Too many grammars implemented in the history of computational linguistics have become obsolete, not only because of their poor maintainability, but also because of the decay of entire software and hardware platforms.

The first measure to be taken against the ”bit rot” of grammars is to write them in **well-defined formats** that can be implemented independently of platform. This requirement is more or less an axiom in programming language development: a language must have syntax and semantics specifications that are independent of its first implementation; otherwise the first implementation risks to remain the only one.

Secondly, since grammar engineering is to a large extent software engineering, grammar formalisms should learn from programming language techniques that have been found useful in this respect. Two such techniques are **static type systems** and **module systems**. Since grammar formalism implementations are mostly descendants of Lisp and Prolog, they usually lack a static type system that finds errors at compile time. In a complex task like grammar writing, compile-time error detection is preferable to run-time debugging whenever possible. As for modularity, traditional grammar formalisms again inherit from Lisp and Prolog low-level mechanisms like macros and file includes, which in modern languages like Java and ML have been replaced by advanced module systems akin in rigour to type systems.

Thirdly, as another lesson from software engineering, grammar writing should permit an increasing use of **libraries**, so that programmers can build on ealier code. Types and modules are essential for the management of libraries. When a new language is developed, an effort is needed in creating libraries for the language, so that programmers can scale up to real-size tasks.

Fourthly, a grammar formalism should have a **stable and efficient implementation** that works on different platforms (hardware and operating systems). Since grammars are often parts of larger language-processing systems (such as translation tools or dialogue systems), their **interoperability** with other components is an important issue. The implementation should provide compilers to standard formats, such as databases and speech recognition language models. In addition to interoperability, such compilers also help keeping the grammars alive even if the original grammar formalism

[*] Now at Google Inc.

Proceedings of the EACL 2009 Demonstrations Session, pages 57–60,
Athens, Greece, 3 April 2009. ©2009 Association for Computational Linguistics

ceases to exist.

Fifthly, grammar formalisms should have **rich documentation**; in particular, they should have accessible tutorials that do not demand the readers to be experts in a linguistic theory or in computer programming. Also the libraries should be documented, preferably by automatically generated documentation in the style of JavaDoc, which is guaranteed to stay up to date.

Last but not least, a grammar formalism, as well its documentation, implementation, and standard libraries, should be **freely available open-source software** that anyone can use, inspect, modify, and improve. In the domain of general-purpose programming, this is yet another growing trend; proprietary languages are being made open-source or at least free of charge.

2 The GF programming language

The development of GF started in 1998 at Xerox Research Centre Europe in Grenoble, within a project entitled ”Multilingual Document Authoring” (Dymetman & al. 2000). Its purpose was to make it productive to build controlled-language translators and multilingual authoring systems, previously produced by hard-coded grammar rules rather than declarative grammar formalisms (Power & Scott 1998). Later, mainly at Chalmers University in Gothenburg, GF developed into a functional programming language inspired by ML and Haskell, with a strict type system and operational semantics specified in (Ranta 2004). A module system was soon added (Ranta 2007), inspired by the parametrized modules of ML and the class inheritance hierarchies of Java, although with multiple inheritance in the style of C++.

Technically, GF falls within the class of so-called Curry-style categorial grammars, inspired by the distinction between tectogrammatical and phenogrammatical structure in (Curry 1963). Thus a GF grammar has an **abstract syntax** defining a system of types and trees (i.e. a free algebra), and a **concrete syntax**, which is a homomorphic mapping from trees to strings and, more generally, to **records** of strings and features. To take a simple example, the NP-VP predication rule, written

```
S ::= NP VP
```

in a context-free notation, becomes in GF a pair of an abstract and a concrete syntax rule,

```
fun Pred : NP -> VP -> S
lin Pred np vp = np ++ vp
```

The keyword `fun` stands for function declaration (declaring the function `Pred` of type `NP -> VP -> S`), whereas `lin` stands for linearization (saying that trees of form `Pred np vp` are converted to strings where the linearization of `np` is followed by the linearization of `vp`). The arrow `->` is the normal function type arrow of programming languages, and ++ is concatenation.

Patterns more complex than string concatenation can be used in linearizations of the same predication trees as the rule above. Thus **agreement** can be expressed by using features passed from the noun phrase to the verb phrase. The noun phrase is here defined as not just a string, but as a record with two fields—a string `s` and an agreement feature `a`. Verb-subject inversion can be expressed by making VP into a **discontinuous constituent**, i.e. a record with separate verb and complement fields `v` and `c`. Combining these two phenomena, we write

```
vp.v ! np.a ++ np.s ++ vp.c
```

(For the details of the notation, we refer to documentation on the GF web page.) Generalizing strings into richer data structures makes it smooth to deal accurately with complexities such as German constituent order and Romance clitics, while maintaining the simple tree structure defined by the abstract syntax of `Pred`.

Separating abstract and concrete syntax makes it possible to write **multilingual grammars**, where one abstract syntax is equipped with several concrete syntaxes. Thus different string configurations can be mapped into the same abstract syntax trees. For instance, the distinction between SVO and VSO languages can be ignored on the abstract level, and so can all other $\{S,V,O\}$ patterns as well. Also the differences in feature systems can be abstracted away from. For instance, agreement features in English are much simpler than in Arabic; yet the same abstract syntax can be used.

Since concrete syntax is reversible between linearization and parsing (Ljunglöf 2004), multilingual grammars can be used for **translation**, where the abstract syntax works as interlingua. Experience from translation projects (e.g. Burke and Johannisson 2005, Caprotti 2006) has shown that the interlingua-based translation provided by GF gives good quality in domain-specific tasks. However, GF also supports the use of a transfer component if the compositional method implied by multilingual grammars does not suffice (Bringert and Ranta

2008). The language-theoretical strenght of GF is between mildly and fully context-sensitive, with polynomial parsing complexity (Ljunglöf 2004).

In addition to multilingual grammars, GF is usable for more traditional, large-scale unilingual grammar development. The ”middle-scale” resource grammars can be extended to wide-coverage grammars, by adding a few rules and a large lexicon. GF provides powerful tools for building morphological lexica and exporting them to other formats, including Xerox finite state tools (Beesley and Karttunen 2003) and SQL databases (Forsberg and Ranta 2004). Some large lexica have been ported to the GF format from freely available sources for Bulgarian, English, Finnish, Hindi, and Swedish, comprising up to 70,000 lemmas and over two million word forms.

3 The GF Resource Grammar Library

The GF Resource Grammar Library is a comprehensive multilingual grammar currently implemented for 12 languages: Bulgarian, Catalan, Danish, English, Finnish, French, German, Italian, Norwegian, Russian, Spanish, and Swedish. Work is in progress on Arabic, Hindi/Urdu, Latin, Polish, Romanian, and Thai. The library is an open-source project, which constantly attracts new contributions.

The library can be seen as an experiment on how far the notion of multilingual grammars extends and how GF scales up to wide-coverage grammars. Its primary purpose, however, is to provide a programming resource similar to the standard libraries of various programming languages. When all linguistic details are taken into account, grammar writing is an expert programming task, and the library aims to make this expertise available to non-expert application programmers.

The coverage of the library is comparable to the Core Language Engine (Rayner & al. 2000). It has been developed and tested in applications ranging from a translation system for software specifications (Burke and Johannisson 2005) to in-car dialogue systems (Perera and Ranta 2007).

The use of a grammar as a library is made possible by the type and module system of GF (Ranta 2007). What is more, the API (Application Programmer’s Interface) of the library is to a large extent language-independent. For instance, an NP-VP predication rule is available for all languages, even though the underlying details of predication vary greatly from one language to another.

A typical domain grammar, such as the one in Perera and Ranta (2007), has 100–200 syntactic combinations and a lexicon of a few hundred lemmas. Building the syntax with the help of the library is a matter of a few working days. Once it is built for one language, porting it to other languages mainly requires writing the lexicon. By the use of the inflection libraries, this is a matter of hours. Thus porting a domain grammar to a new language requires very effort and also very little linguistic knowledge: it is expertise of the application domain and its terminology that is needed.

4 The GF grammar compiler

The GF grammar compiler is usable in two ways: in batch mode, and as an interactive shell. The shell is a useful tool for developers as it provides testing facilities such as parsing, linerization, random generation, and grammar statistics. Both modes use PGF, **Portable Grammar Format**, which is the ”machine language” of GF permitting fast run-time linearization and parsing (Angelov & al. 2008). PGF interpreters have been written in C++, Java, and Haskell, permitting an easy embedding of grammars in systems written in these languages. PGF can moreover be translated to other formats, including language models for speech recognition (e.g. Nuance and HTK; see Bringert 2007a), VoiceXML (Bringert 2007b), and JavaScript (Meza Moreno and Bringert 2008). The grammar compiler is heavily optimizing, so that the use of a large library grammar in small run-time applications produces no penalty.

For the working grammarian, static type checking is maybe the most unique feature of the GF grammar compiler. Type checking does not only detect errors in grammars. It also enables aggressive optimizations (type-driven partial evaluation), and **overloading resolution**, which makes it possible to use the same name for different functions whose types are different.

5 Related work

As a grammar development system, GF is comparable to Regulus (Rayner 2006), LKB (Copestake 2002), and XLE (Kaplan and Maxwell 2007). The unique features of GF are its type and module system, support for multilingual grammars, the large number of back-end formats, and the availability of libraries for 12 languages. Regulus has resource grammars for 7 languages, but they are smaller in scope. In LKB, the LinGO grammar matrix has been developed for several languages (Bender and Flickinger 2005), and in XLE, the Pargram grammar set (Butt & al. 2002). LKB and XLE tools have been targeted to linguists working with large-scale grammars, rather than for general programmers working with applications.

References

[Angelov et al.2008] K. Angelov, B. Bringert, and A. Ranta. 2008. PGF: A Portable Run-Time Format for Type-Theoretical Grammars. Chalmers University. Submitted for publication.

[Beesley and Karttunen2003] K. Beesley and L. Karttunen. 2003. *Finite State Morphology*. CSLI Publications.

[Bender and Flickinger2005] Emily M. Bender and Dan Flickinger. 2005. Rapid prototyping of scalable grammars: Towards modularity in extensions to a language-independent core. In *Proceedings of the 2nd International Joint Conference on Natural Language Processing IJCNLP-05 (Posters/Demos)*, Jeju Island, Korea.

[Bringert and Ranta2008] B. Bringert and A. Ranta. 2008. A Pattern for Almost Compositional Functions. *The Journal of Functional Programming*, 18(5–6):567–598.

[Bringert2007a] B. Bringert. 2007a. Speech Recognition Grammar Compilation in Grammatical Framework. In *SPEECHGRAM 2007: ACL Workshop on Grammar-Based Approaches to Spoken Language Processing, June 29, 2007, Prague*.

[Bringert2007b] Björn Bringert. 2007b. 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.

[Bringert2008] B. Bringert. 2008. Semantics of the GF Resource Grammar Library. Report, Chalmers University.

[Burke and Johannisson2005] D. A. Burke and K. Johannisson. 2005. Translating Formal Software Specifications to Natural Language / A Grammar-Based Approach. In P. Blache and E. Stabler and J. Busquets and R. Moot, editor, *Logical Aspects of Computational Linguistics (LACL 2005)*, volume 3492 of *LNCS/LNAI*, pages 51–66. Springer.

[Butt et al.2002] M. Butt, H. Dyvik, T. Holloway King, H. Masuichi, and C. Rohrer. 2002. The Parallel Grammar Project. In *COLING 2002, Workshop on Grammar Engineering and Evaluation*, pages 1–7. URL

[Caprotti2006] O. Caprotti. 2006. WebALT! Deliver Mathematics Everywhere. In *Proceedings of SITE 2006. Orlando March 20-24*.

[Copestake2002] A. Copestake. 2002. *Implementing Typed Feature Structure Grammars*. CSLI Publications.

[Curry1963] H. B. Curry. 1963. Some logical aspects of grammatical structure. In Roman Jakobson, editor, *Structure of Language and its Mathematical Aspects: Proceedings of the Twelfth Symposium in Applied Mathematics*, pages 56–68. American Mathematical Society.

[Dymetman et al.2000] M. Dymetman, V. Lux, and A. Ranta. 2000. XML and multilingual document authoring: Convergent trends. In *COLING, Saarbrücken, Germany*, pages 243–249.

[Forsberg and Ranta2004] M. Forsberg and A. Ranta. 2004. Functional Morphology. In *ICFP 2004, Showbird, Utah*, pages 213–223.

[Ljunglöf2004] P. Ljunglöf. 2004. *The Expressivity and Complexity of Grammatical Framework*. Ph.D. thesis, Dept. of Computing Science, Chalmers University of Technology and Gothenburg University.

[Meza Moreno and Bringert2008] M. S. Meza Moreno and B. Bringert. 2008. Interactive Multilingual Web Applications with Grammarical Framework. In B. Nordström and A. Ranta, editors, *Advances in Natural Language Processing (GoTAL 2008)*, volume 5221 of *LNCS/LNAI*, pages 336–347.

[Perera and Ranta2007] N. Perera and A. Ranta. 2007. Dialogue System Localization with the GF Resource Grammar Library. In *SPEECHGRAM 2007: ACL Workshop on Grammar-Based Approaches to Spoken Language Processing, June 29, 2007, Prague*.

[Power and Scott1998] R. Power and D. Scott. 1998. Multilingual authoring using feedback texts. In *COLING-ACL*.

[Ranta2004] A. Ranta. 2004. Grammatical Framework: A Type-Theoretical Grammar Formalism. *The Journal of Functional Programming*, 14(2):145–189.

[Ranta2007] A. Ranta. 2007. Modular Grammar Engineering in GF. *Research on Language and Computation*, 5:133–158.

[Rayner et al.2000] M. Rayner, D. Carter, P. Bouillon, V. Digalakis, and M. Wirén. 2000. *The Spoken Language Translator*. Cambridge University Press, Cambridge.

[Rayner et al.2006] M. Rayner, B. A. Hockey, and P. Bouillon. 2006. *Putting Linguistics into Speech Recognition: The Regulus Grammar Compiler*. CSLI Publications.