Grammars as Software Libraries

Aarne Ranta
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

à la mémoire de Gilles Kahn

Abstract

Grammars of natural languages are needed in programs like natural language interfaces and dialogue systems, but also more generally, in software localization. Writing grammar implementations is a highly specialized task. For various reasons, no libraries have been available to ease this task. This paper shows how grammar libraries can be written in GF (Grammatical Framework), focusing on the software engineering aspects rather than the linguistic aspects. As an implementation of the approach, the GF Resource Grammar Library currently comprises ten languages. As an application, a translation system from formalized mathematics to text in three languages is outlined.

1 Introduction

How can we generate natural language text from a formal specification of meaning, such as a formal proof? Coscoy, Kahn and Théry [10] studied the problem and built a program that worked for all proofs constructed in the Coq proof assistant [30]. Their program translates structural text components, such as *we conclude that*, but leaves propositions expressed in formal language:

```
We conclude that Even(n) -> Odd(Succ(n)).
```

A similar decision is made in Isar [32], whereas Mizar [31] permits English-like expressions for some predicates. One reason for stopping at this level is certainly that typical users of proof systems are comfortable with reading logical formulas, so that only the proof-level formalization needs translation.

Another good reason for not translating propositions in a system like [10] is the difficulty of the task. It is enough to look at any precise formalization of natural language syntax to conclude that a lot of work and linguistic knowledge is demanded to get it right. This knowledge is largely independent of the domain of translations: the very same grammatical problems appear in tasks as different from proofs as, for instance, systems for hands-free controling of an MP3 player in a car [23].

In this paper, we will introduce a (to our knowledge) novel approach to natural language programming tasks. We view grammar rules as specialist knowledge, which should be encapsulated in **libraries**. Using a grammar in an application program then becomes similar to, for instance, using a numerical analysis library in a graphics rendering programme. The user of the library just has to specify on a high abstraction level what she wants—for instance, that she wants to build a sentence from a certain noun phrase and a certain adjective. The library takes care of picking the proper forms of words (which e.g. in French must obey the rules of gender and number agreement) and putting the words in right order (which e.g. in German depends on whether the sentence is subordinate or a main clause).

To introduce the grammar-related problems in programming, we will start with a simple example from the area of **software localization**—a task of natural language rendering of messages produced by a program. We continue with an outline of GF (Grammatical Framework, [25]), which is a special-purpose programming language for writing grammars, in particular designed to permit modularity and information hiding [26]. One major asset of GF is a Resource Grammar Library, which formalizes the central grammatical structures of 14 languages [27]. We give an outline of this library, with the main focus on the organization, presentation, and evaluation of the library from the software engineering point of view (rather than the linguistic point of view, which is treated in other publications). As an example of the use of the library, we show how to use the library for the translation of formalized mathematics to natural language, which can be seen as complementing the work of [10].

2 Grammars and software localization

2.1 A very simple example

Many programs produce natural-language output in the form of short messages. In software localization, these messages must be translated to new languages. One important property of translation is grammatical correctness. Even in English, this is often violated: an email program may tell you,

```
You have 1 messages
```

If a little more thought has been put into the program, it might say

```
You have 1 message(s)
```

The code that should be written to get the grammar right is of course

```
msgs n = "You have" ++ show n ++ messages
  where
    messages = if n==1 then "message" else "messages"
```

Now, what is it we need in order to generate this in a new language?

First of all, we have to know the words *you*, *have*, and *message* in the new language. This is not quite as simple as it may sound. For instance, looking up the word *message* in an English-Swedish dictionary on the web[1] gives the variants *bud* and *budskap*. Which one to choose? In fact, the correct answer is neither: the translation of *message* in the domain of emails is *meddelande*.

In addition to the dictionary forms of the words, we need to know how they are inflected. Only when we know that the plural of *meddelande* is *meddelanden*, can we write the Swedish rule

```
msgs n = "Du har" ++ show n ++ messages
  where
    if n == 1 then "meddelande" else "meddelanden"
```

However, it is not universal that only the one/many distinction affects inflection. In Arabic, there are five cases [13]:

```
if n ==  1 then "risAlatun" else
if n ==  2 then "risAlatAni" else
if n <  11 then "rasA'ila" else
if n % 100 == 0 then "risAlatin" else
                     "risAlatan"
```

From these strictly grammatical decisions we arrive to more pragmatic, or cultural, ones. In many languages, we have to know the proper way to politely address the user. In Swedish, one tends to use the familiar *du har 3 meddelanden* rather than the formal *ni har 3 meddelanden*. In French, the preference is the opposite: *vous avez 3 messages* rather than *tu as 3 messages*. The preferred choice depends not only on language but also on the intended audience of the program.

Localization clearly needs more knowledge than what can be found in a dictionary. Hiring native-speaker programmers is one solution, but only if these programmers have explicit grammatical knowledge of their native languages. In general, the expertise that is needed is the linguist's expertise. At the same time, localization may require expertise on the application domain rather than in linguistics, so that truly appropriate terminology is used. Can we find these two experts in one and the same person? Or do we have to hire two programmers per language?

2.2 A library-based solution

A common technique used in localization are databases that contain words and standard texts in different languages. Such a library may contain a key, `YouHaveMessages`, or simply, `sentence_2019`, which is rendered as a function of language. Companies may have databases of thousands of sentences used for localizing their products.

[1] `www.freedict.com/onldict/swe.html`

If a localization library is really sophisticated, the renderings are not just constant strings but can be templates, so that e.g. `YouHaveMessages` is a template in which a number is filled and the proper rendering is chosen by case analysis on the number.

Now, an unlimited number of sentences of the same form as *you have 3 messages* can be produced by changing the subject and the object:

```
You have 4 points.
We have 2 cases.
```

The first example could be from a game, whereas the second appears in the proof renderings of [10]. To cover all possible variations, a database is not enough: something like a **grammar** is needed. However, grammars as usually written by linguists do not have a format that is usable for this task. For instance, a context-free grammar or a unification grammar defines the set of strings of a language, but it does not provide explicit functions for rendering particular structures.

To see what format is required of a grammar, let us take a cue from databases with keys to ready-made sentences and templates. Keys to sentences can be seen as constants, and keys to templates as functions:

```
Hello : Sentence
YouHaveMessages : Number -> Sentence
```

A grammar arises from this picture in a natural way: we just add more types of expressions and more complex functions, including recursive ones:

```
Modify : Adjective -> Noun -> Noun
```

Following a tradition in grammar, we call these types the **categories** of the grammar. The "keys" could be called **grammatical functions**. The categories and the function type signatures together form the **API** (Application Programmer's Interface) of the grammar library: they are everything the user of the library needs in order to build grammatically correct expressions. In addition, the library has to provide a **rendering function**, such that for each category `C`,

```
render : Language -> C -> String
```

The linguistic knowledge contained in the library is hidden behind the API showing the categories, the grammatical functions, and the rendering functions; the user of the library does not need to care about how they are implemented. Notice, moreover, that the API is independent of rendering language: the same combinations of grammatical functions can be rendered to different languages by varying the `Language` parameter.

Returning to the *n messages* example, we would need a grammar library API containing the categories

```
Sentence, NounPhrase, Noun, Number
```

and the constructor functions

```
Have     : NounPhrase -> NounPhrase -> Sentence
NumberOf : Number -> Noun -> NounPhrase

PoliteYou, FamiliarYou, We : NounPhrase
Message, Point, Case : Noun
```

Then we can translate the examples above by using different values of `lang` in

```
render lang (Have PoliteYou   (NumberOf 1 Message))
render lang (Have FamiliarYou (NumberOf 4 Point))
render lang (Have We          (NumberOf 3 Case))
```

2.3 Searching translations by parsing

If localization is implemented with an ordinary database, we can use a string in one language to search translations in other languages. In a grammar, the corresponding technique is **parsing**, i.e. the inverse of the rendering function.

```
parse : Language -> String -> C
```

This would enable us to write

```
msgs lang n = render lang (parse english "you have n messages")
```

thus avoiding the manual construction of grammatical function applications. This can be a very efficient way to use a grammar library. However, since natural languages are ambiguous, `parse` may give many results:

```
"you have 3 messages"

Have PoliteYou   (NumberOf 3 Message)
Have FamiliarYou (NumberOf 3 Message)
Have PluralYou   (NumberOf 3 Message)
```

It then remains to the user of the library to select the correct alternative, and she must thus have at least some understanding of the grammatical functions.

3 Implementing a grammar library in GF

Those who know GF [25] must have recognized the introduction as a seduction argument eventually leading to GF. The main components of a grammar library correspond exactly to the main components of GF:

- categories and grammatical functions = **abstract syntax**
- rendering and parsing = **concrete syntax**
- abstract library objects = **abstract syntax trees**

We refer to [25, 26, 27] for the details of GF. Let us just show a set of GF modules forming a very simple English-French language library. First, there is an abstract syntax module listing categories (`cat`) and grammatical functions (`fun`):

```
abstract Lib = {
  cat
    Noun ;
    Adjective ;
  fun
    Modify : Adjective -> Noun -> Noun ;
}
```

This module has adjectives and nouns, and a function that modifies a noun by an adjective (e.g. *even number*, *nombre pair*).

Second, there is a concrete syntax module `of` this abstract syntax, assigning a **linearization type** (`lincat`) to each category, and a **linearization function** (`lin`) to each function. Linearization may involve **parameters** (`param`) that control the rendering of abstract syntax trees.

```
concrete LibEng of Lib = {
  lincat
    Noun = {s : Number => Str} ;
    Adjective = {s : Str} ;
  lin
    Modify adj noun = {
      s = table {n => adj.s ++ noun.s ! n}
    } ;
  param
    Number = Singular | Plural ;
}
```

Linearization is a homomorphism that obeys the linearization types. Linearization types, in general, are record types that contain all relevant linguistic information. Minimally, they contain just a string, as `Adjective` does in this English example. But `Noun` in this example has a **table** (a **finite function**), which produces a string as a function of `Number` (defined by the constructors `Singular` and `Plural`). The linearizarion of modification passes the number parameter (variable `n`) to the noun (where `!` marks selection from a table) and concatenates (`++`) the adjective with the resulting noun form. In this way, we get (*one*) *new message* and (*three*) *new messages*

In the French module, we have different linearization types and different word order.

```
concrete LibFre of Lib = {
  lincat
    Noun = {s : Number => Str ; g : Gender} ;
    Adjective = {s : Gender => Number => Str ; isPre : Bool} ;
  lin
    Modify adj noun = {
      s = table {n => case adj.isPre of {
              True  => adj.s ! noun.g ! n ++ noun.s ! n ;
              False => noun.s ! n ++ adj.s ! noun.g ! n
              }
            } ;
      g = noun.g
    } ;
  param
    Number = Singular | Plural ;
    Gender = Masculine | Feminine ;
    Bool = True | False ;
}
```

The modification rule shows **agreement** between the noun and the adjective: the adjective is inflected by selecting the gender of the noun. In this way, we get *nombre pair* but *somme paire*. Unlike in English, the adjective can be placed after the noun; a boolean parameter is used to take care of whether the adjective is placed before of after the noun (e.g. *nouveau message*, "new message" vs. *message privé*, "private message").

This simple example is enough to show that natural languages have complex grammars but also, more importantly, that the abstract syntax can abstract away from complexities like word order and agreement, which are different from one language to another. An application programmer using the library can thus be freed from thinking about these details. With the adjective `new` and the noun `message` added to the abstract and the concrete syntaxes, the abstract syntax tree `Modify new message` will produce all desired forms of this combination in both languages.

The example we just showed is a a **multilingual grammar**: an abstract syntax together with a set of concrete syntaxes. A multilingual grammar can be used as a translator, where translation is the composition of parsing from one language with linearization to another. But it is also a suitable format for a grammar library. As the API of the library, the abstract syntax can be used, together with the names of the concrete syntaxes, showing what languages are available.

The `render` method is a direct application of the linearization rules. Also a `parse` method is available, as a consequence of the **reversibility** property of GF grammars: a set of linearization rules automatically generates a parser.

3.1 Using GF grammars in other programming languages

A GF grammar library can obviously be used for writing GF grammars—but this is not yet very useful if the grammar is to be a part of an application written in another programming languages. The simplest way to make it usable is off-line: to construct texts that are cut and pasted to the application, or stored in databases of fixed phrases. But this is not always possible, since the proper natural-language output of a program may depend on the run-time input of the program, as was the case in the *n messages* example, and also in applications like natural-language interfaces to proof assistants.

A fully satisfactory method is to compile the multilingual grammar into code usable in other languages. These languages can then make the rendering functions available as ordinary functions. For instance, in Java 1.5, we have

```
String linearize(Language l, Tree t)
Collection<Tree> parse(Language l, Category c, String str)
```

(see [5]). In order to use these functions, the programmer also has to be able to construct and analyse objects of type `Tree`. This can be done in two ways. The first alternative is a universal type of trees. This type has a constructor that builds a tree from a label representing an abstract syntax function, and an array of subtrees:

```
Tree(String label, Tree[] children)
```

While this tree type is powerful enough to represent all trees, it also permits false ones: the types of the abstract syntax functions do not constrain the construction of trees. A better alternative is to encode the abstract syntax types by using Java's class system, in the same way as Appel [1] does for the abstract syntax of programming languages. In this method, an abstract base class is created for every category of abstract syntax. Every function is encoded as a class extending its value category:

```
public abstract class Noun { ... }
public abstract class Adjective  { ... }
public class Modify extends Noun {
  public final Adjective adjective_ ;
  public final Noun noun_ ;
  public Modify(Adjective p1, Noun p2){ ... }
}
```

The concrete syntax implementation is not presented to the application programmer, and is in that sense uninteresting. But how does it work? The method we have used is to compile GF grammars into a simpler low-level format called PGF, Portable Grammar Format. This format consists essentially of arrays of strings and pointers to other arrays, implementing the linearization records of all well-typed trees. While the compiler from GF to PGF is a highly complex program, a PGF interpreter is fairly easy to write in any programming language. Such interpreters have been written in Java, Haskell, C++, and Prolog [27]. Also tree building is supported, so that e.g. a Java class system in the format shown above can be generated from an abstract syntax.

An alternative to the PGF interpreter is compilation from PGF to host program code. This method is useful if high performance is needed. For instance, in order to use GF grammars in portable devices with severe constraints, we have developed a compiler from PGF to C. In order to run language processing in web browsers, we compile PGF to JavaScript.

3.2 The GF Resource Grammar Library

By building the high-level grammar formalism GF and its compiler to the low-level format PGF, we have created two components of the infrastructure for grammar engineering. The next component, which we now want to focus on, is the **GF Resource Grammar Library** [27, 19]. The library project started in 2001, with the purpose of creating a standard library for GF, and thereby to make GF more usable for non-linguist programmers. As of version 1.2 (December 2007), the library covers ten languages: Danish, English, Finnish, French, German, Italian, Norwegian, Russian, Spanish, and Swedish. For these languages, a full coverage of the API is provided. Partial but already useful implementations exist for Arabic [14], Bulgarian, Catalan, and Urdu [18].

The GF Resource Grammar Library has had three major applications outside the group developing the library: in the KeY project, for translating between formal and informal software specifications [2, 6]; in the WebALT project, for translating mathematical exercises from formal representations to seven languages [8]; and in the TALK project, for localizing spoken dialogue systems to different languages [23]. These projects together have forced the library to cover both written technical language and spoken casual language. They have also involved library users with varying backgrounds: linguists, computer scientists, mathematicians, engineers.

In the following sections, we will discuss some critical questions of grammar libraries, including design, presentation, and programming language aspects. We will do this in the light of the experience gained from the GF Resource Grammar Library project.

4 Design questions for a grammar library

When designing a library, the first question is of course *coverage*. A grammar library should cover the grammars of the involved languages—but what this means is not so clear-cut for natural languages as it is for programming languages. Natural languages are open-ended: they evolve all the time, and the boundary between what is grammatical and what is not is often vague.

The second question is *organization and presentation*: division into modules, level of granularity, orthogonality. In the domain of grammars, we have also the choice between traditional "school grammar" concepts (which however do not suffice for all purposes) and modern, sophisticated linguistic concepts. (which however vary from one linguistic theory to another).

While these design questions have analogues in other areas of software libraries, the use of grammars as libraries has not been studied before. The reason is that grammars have mainly been written to provide stand-alone parsers and generators, with, so to say, `render` and `parse` as the only points of access.

5 The coverage of the library

5.1 Morphology

If we want the library to cover a language, the first thing that we need is **inflectional morphology**. This component enables the analysis and synthesis of all word forms in the language. What the rules of the morphology are is a well understood question for all the languages in GF Resource Grammar Library, so that the implementation is a question of engineering. GF is a typed functional programming language, and it is therefore natural to use similar techniques as in the Zen toolkit [17] and in Functional Morphology [15].

A functional-style morphology implementation consists of an **inflection engine** and a **lexicon**. The inflection engine is a set of **paradigms**, that is, functions that compute a full inflection table from a single word form. The lexicon can then be presented as a list of word-paradigm pairs.

It is the lexicon that makes the morphology usable as a tool for tasks like analysing texts. In a library, however, a static lexicon is less important than the inflection engine. Domain-specific applications of the library often require new words not appearing in any standard lexicon. Fortunately, such technical words tend to follow regular inflection patterns, and it is easy to provide paradigms for defining their inflection without very much knowledge of linguistics. The most useful part of a lexicon provided by a library is one containing irregular words, e.g. the French irregular verbs, as well as function words, such as pronouns, which are both frequent, domain-independent, and very irregular. These lexica together result in some hundreds of words in each language.

5.2 Syntax

The most challenging part of the library is **syntax**. In the GF Resource Grammar Library, we started from an extension of the PTQ fragment of Montague ("Proper Treatment of Quantification in Ordinary English", [21]) and ended up with a coverage similar to the CLE (Core Language Engine [28]). This language fragment has proven sufficient for the aforementioned large-scale applications of GF Resource Grammar Library, as well as for many small-scale test applications.

The syntax in the GF Resource Grammar Library covers the usual syntactic structures within sentences: predication, coordination, relative clauses, indirect questions, embedded sentences, pronouns, determiners, adjectives, etc. It also covers structures above the sentence level: the topmost category is `Text`. Texts are lists of `Phrase`s, which can be declarative sentences, questions, imperatives, and exclamations; also subsentential phrases are covered, as they are needed in dialogues: (*What do you want to drink?*) *Red wine.*

6 The organization and presentation of the library

6.1 Morphology

The morphology API is a set of paradigms. A paradigm takes a string (usually, the dictionary form also known as the **lemma**) and produces a complete inflection table. For instance, French verb paradigms look as follows:

```
v_besch56 : Str -> V  -- mettre
```

The naming of verb paradigms in French and other Romance languages follows the authoritative *Bescherelle* series of books [4]. Each paradigm in the API is endowed with a comment containing some example words.

Traditionally, as in the *Bescherelle* books, an inflectional paradigm is a function that takes one form and produces all the others (51 forms in the case of French verbs). Paradigm are identified by the sets of endings that are used for each word form: if two words show any difference in their endings, they belong to different paradigms. Thus the French *Bescherelle* contains 88 verb paradigms. This is a prohibitive number for the user of the library, and it is easy to make mistakes in picking a paradigm.

Fortunately, most paradigms in most languages are unproductive, in the sense that they apply only to a limited number of words—all new words can be treated with a handful of paradigms, often known as "regular". In *Bescherelle*, for instance, less than 10 of the 88 paradigms are regular in this sense. They are, moreover, predictable: to choose what paradigm to use, it is enough to consider the ending of the verb infinitive form. This prediction can be encoded as a **smart paradigm**, which inspects its argument using regular expression patterns, and dispatches to the productive *Bescherelle* paradigms:

```
mkV : Str -> V = \v ->
  case v of {
    _ + "ir"              => v_besch19 v ;  -- finir
    _ + "re"              => v_besch53 v ;  -- rendre
    _ + "éger"            => v_besch14 v ;  -- assiéger
    _ + ("eler" | "eter") => v_besch12 v ;  -- jeter
    _ + "éder"            => v_besch10 v ;  -- céder
    _ + "cer"             => v_besch7  v ;  -- placer
    _ + "ger"             => v_besch8  v ;  -- manger
    _ + "yer"             => v_besch16 v ;  -- payer
    _                     => v_besch6  v    -- aimer
  } ;
```

The morphology API for French verbs now consists of this smart paradigm, together with a lexicon of those 379 verbs that do not follow it.

The smart paradigm idea was the single most important reason to add regular pattern matching to GF. The GF Resource Grammar Library has shown that it scales up well. For instance, in Finnish, which (together with Arabic) is morphologically the most complex of the library languages, 90% of words are covered by one-argument smart paradigms.

6.2 Syntax

Maybe the most useful and at the same time the most surprising feature of the GF Resource Grammar Library is that its syntax API is language-independent. In other words, the library describes all languages as having the same structure. To make this possible, the power of separating concrete syntax from abstract syntax is sometimes stretched to the extreme. However, since linearization is a homomorphism, the relation between trees and their linearizations remains compositional.

In version 1.2, the syntax API consists of an abstract syntax with 44 categories and 190 functions. If each language had a separate API, the library would have something like 440 categories and 1900 functions. The common syntax API thus improves the manageability of the library by an order of magnitude. Another advantage is that, once a programmer learns to use the library for one language, she can use it for all the other languages as well.

The disadvantages are mostly on the implementor's side: the concrete syntax is often more complex and less natural than it would be if each language had its own abstract syntax. The user of the library will mainly notice this disadvantage as long and memory-demanding compilations from time to time.

The common API is complete and sound in the sense that it permits the programmer to express everything in all languages with grammatically correct sentences. The resulting language is highly normalized and can be unidiomatic. Therefore the library also provides language-dependent syntax extensions. For instance, the great number of tenses in Romance languages (e.g. simple vs. composite past) has no counterpart in other languages. Just some of the tenses are accessible via the common API, and the rest via language-dependent extensions.

6.3 The common syntax API

The concepts of syntax are more abstract and less commonly known than the concepts of morphology. For instance, how many non-linguist programmers would know that a sentence is built from a **noun phrase** and a **verb phrases**, and that a verb phrase is formed by combining a verb with a sequence of **complements** that depend on the **subcategorization frame** of the verb?

Let us see how the library deals with sentence construction. There are several categories of verbs, corresponding to the different subcategorization frames. The API shows the category declarations, each commented with a description and an example. Here are some of the frames we use:

```
V    -- one-place verb             e.g. "sleep"
V2   -- two-place verb             e.g. "love"
V3   -- three-place verb           e.g. "show"
```

```
  VS   -- sentence-complement verb     e.g. "claim"
```

For each category, there is a function that forms verb phrases:

```
  UseV    : V   -> VP               -- sleep
  ComplV2 : V2  -> NP -> VP         -- use it
  ComplV3 : V3  -> NP -> NP -> VP   -- send it to her
  ComplVS : VS  -> S  -> VP         -- know that she runs
```

Verb phrases can moreover be formed from the copula (*be* in English) with different kinds of complements,

```
  UseComp : Comp -> VP              -- be warm
  CompAP  : AP   -> Comp            -- (be) small
  CompNP  : NP   -> Comp            -- (be) a soldier
  CompAdv : Adv  -> Comp            -- (be) here
```

These complements can again be built in many ways; for instance, an adjectival phrase (`AP`) can be built as the positive form of an the adjective (`A`), using the function

```
  PositA  : A -> AP                 -- warm
```

On the top level, a **clause** (`Cl`) can be built by putting together a noun phrase and a verb phase:

```
  PredVP  : NP -> VP -> Cl          -- this is warm
```

A clause is like a **sentence** (`S`), but with unspecified tense, anteriority (to do vs. to have done), and polarity (positive or negative). A complete sentence is thus produced by fixing these features:

```
  UseCl   : Cl -> Tense -> Ant -> Pol -> S
```

The structure of this abstract syntax is motivated by the demands of completeness, succinctness, and non-redundancy. Similar ways of forming trees are factored out. The categories `VP` and `Comp` are examples of this: in a sense, they are artifacts of the linguistic theory.

6.4 Ground API vs. end-user API

In the common syntax API, trees become deeply hierarchical. The very simple sentence *this is warm* becomes

```
  UseCl TPres ASimul PPos
    (PredVP this_NP (UseComp (CompAP (PositA (regA "warm")))))
```

Factoring out similar structures, as succinctness and non-redundancy in general, is good for the implementors of the resource grammars, since it minimizes the duplication of work. For users, however, constructing deeply nested trees, and even reading them, is intimidating, especially because it is difficult to make the naming conventions completely logical and easy to remember.

After the completion of the first version of GF Resource Grammar Library, the main attention was devoted to making the library easier to understand. The succinct, non-redundant API as described above is now called the **ground API**. Its main purpose is to serve as a specification of the functionalities that a resource implementation must provide. The **end-user API** has different requirements. For instance, redundancy can be helpful to the user. The resulting presentation of the end-user API was inspired by one of the most influential software libraries, the Standard Template Library (STL) of C++ [29].

6.5 Overloading

An important instrument in STL is **overloading**: the use of a common name for different functions that in some sense perform similar tasks. Two different criteria of similarity used in STL have proven useful in our library:

- functions that construct objects of the same type: **constructors** in STL
- functions that operate in the same way on their arguments, irrespectively to type: **algorithms** in STL

Overloading in GF is implemented by a compile-time resolution algorithm. As in C++ (and ALGOL68), it performs bottom-up type inference from the arguments. Unlike C++, GF has partial applications, and must therefore use the value type in resolution, in addition to argument types.

The implementation of overloading was unexpectedly simple and efficient. In conclusion, overloading is a small language feature from the implementation point of view but has a deep effect on the way in which the language is used.

6.6 Constructors

What is a constructor? In the technical sense of type theory, *all* functions of an abstract syntax are constructors, since they define the data forms of an inductive system of data types [20]. Overloading of constructors in this sense poses technical problems, for instance, when constructors are used as patterns in case analysis.

However, what corresponds to type-theoretical constructors in C++ is not the C++ constructor functions, but the forms of records of class variables. These variables are usually *not* disclosed to the user of a class. The very purpose of the C++ constructors is to hide the real data constructors. This distinction gives one more reason for GF to have an end-user API separate from the ground API: to achieve full data abstraction, the ground API must not be disclosed to users at all!

We have built a set of overloaded constuctors that covers the resource grammar syntax with just 22 function names. All these names have the form `mk`C, where C is the value category of the constructor. For instance, we have a set of `Cl` forming functions,

```
mkCl : NP -> V   -> Cl                -- he sleeps
mkCl : NP -> V2  -> NP -> Cl          -- he uses it
mkCl : NP -> V3  -> NP -> NP -> Cl    -- he sends it to her
mkCl : NP -> VS  -> S  -> Cl          -- he knows that she runs
mkCl : NP -> A   -> Cl                -- he is warm
mkCl : NP -> AP  -> Cl                -- he is warmer than you
mkCl : NP -> NP  -> Cl                -- he is a man
mkCl : NP -> Adv -> Cl                -- he is here
```

Notice that the set of constructors also flattens the structure of the ground API: the theoretical categories VP and Comp have been eliminated in favor of functions using their constituents directly. Furthermore, we have added a flattening constructor taking an adjective (A), which is a special case of the adjectival phrase (AP).

The constructor forming sentences from clauses shows yet another typical use of overloading: **default arguments**. Any of the arguments for tense, anteriority, and polarity may be omitted, in which case the defaults present, simultaneous, and positive are used, respectively. The API shows optional argument types in parentheses (for the GF compiler, parentheses have only their usual grouping function; the following is really implemented as a group of overloaded functions):

```
mkS : Cl -> (Tense) -> (Ant) -> (Pol) -> S
```

With this set of constructors, the sentence *this is warm* can now be written

```
mkS (mkCl this_NP (regA "warm"))
```

6.7 Combinators

What about the "algorithms" of STL? In grammar, one analogue are grammatical functions that operate on different categories, such as **coordination** (forming conjunctions and disjunctions). These functions are of course redundant because constructors cover the same ground, but often combinators give a more direct and intuitive access to the resource grammar. Here is an example, a part of the coordination function:

```
coord : Conj  -> ListAdv    -> Adv  -- here, now and fast
coord : Conj  -> ListAP     -> AP   -- warm and very tasty
coord : Conj  -> ListNP     -> NP   -- John, Mary and I
coord : Conj  -> ListS      -> S    -- I sleep and you walk
coord : Conj  -> Adv -> Adv -> Adv  -- here and now
coord : DConj -> Adv -> Adv -> Adv  -- both here and now
```

Altogether, there are 4*2*2 = 16 coordination functions, since there are 4 categories that can be coordinated, 2 kinds of coodinated collections (lists of arbitrary length and the special case of two elements), and 2 kinds of coordinating conjunctions (types *and* and *both-and*). Since all of them have different types, they can be represented by one overloaded constant.

7 Success criteria and evaluation

Natural-language grammars are usually evaluated by testing them against some text corpus, such as the Wall Street Journal corpus. The current GF Resource Grammar Library would not perform well in such a test. Even if it did, the test would miss the most important points about the use of the grammar as a library, rather than as a stand-alone parser. Thus we need some new criteria for grammar evaluation, to a large extent similar to software libraries in general.

7.1 The success criteria

Correctness. This is the most important property of any library. The user of the library must be able to rely on the expertise of its authors, and all library functions must thus do the right thing. In the case of a grammar library, this means **grammatical correctness** of everything that is type-correct in the grammar.

Coverage. In tasks such as localization or document generation, this means above all **semantic coverage**: although limited, the language fragment must be sufficient for expressing whatever programmers need to express.

Usability. In this case, usability by non-linguists is the interesting question. The success in this point depends on presentation and documentation, rather than on the implementation.

Efficiency. This is a property often mentioned in the C++ community: using the library should not create any run-time overhead compared with hand-written code [29]. Interestingly, a considerable amount of compile-time overhead is created in both C++ (because of template instantiation) and GF (because of partial evaluation; see [25]).

7.2 These are not our success criteria

Completeness, in the sense of the grammar's being able to parse all expressions. While this would be useful in applications that need to parse user input, it would clutter the grammar with questionable constructions and compromise its correctness.

Semantic correctness, in the sense of the grammar only producing meaningful expressions. Now we can produce grammatically well-formed nonsense:

```
colourless green ideas sleep furiously
draw an equilateral line through the rectangular circle
```

It is difficult to rule out such sentences by general principles. The philosophy of the resource grammar library is that semantics is given in applications, not in the library. What is meaningful, and what meaning is, varies from one domain to the other.

Translation equivalence. The common syntax API does not guarantee common meanings, let alone common pragmatic value. In applications, it is often necessary to use the API in different ways for different languages. For instance, a standard English mathematical exercise uses the imperative, whereas French uses the infinitive:

```
Compute the sum X.   -- *Calculez la somme X.
Calculer la somme X. -- *To compute the sum X.
```

It is the programmer of the mathematics application who selects the right constructions in both languages. The grammar library only takes care of the proper renderings of these constructions. Of course, a special-purpose mathematics library can introduce a function for constructing exercises, which uses the ground library in different ways for English and French.

Linguistic innovation. The idea of the resource grammar library is to formalize a body of "known facts" about languages and offer them to non-linguist users. While we do believe that some innovation was needed to make the library work, the purpose of the API is to hide this from the users, and make it look natural and easy.

7.3 Evaluation

To evaluate correctness, coverage, and usability, it is good to have applications from different areas and by different users. In this sense, testing a grammar library is much like testing any software. An extra method for testing grammatical correctness is the possibility of automatically generating trees and linearizing them, and inspecting the results. Version 1.0 of the GF Resource Grammar Library was released when a stable point was reached in this process.

Completeness in the usual mathematical sense is even possible to prove, by writing a translator from a system of logic to the library API. This has been done for several systems, thus proving instances of expressive completeness.

Compile-time efficiency is a serious problem with some resource grammars (in particular, the Romance languages and Finnish). This has led us to looking for improvements to the partial evaluation method used. But grammars written by using resource grammars as libraries, when compiled, are free from overhead, because the partial evaluation specializes them accurately to the type system of the application, as shown in [25].

8 Example: rendering mathematical concepts

Mathematical text is a mixture of natural language and symbolic notation. While the proportion of symbolic notation varies, the following rules are usually obeyed:

- Only logically atomic sentences are symbolic: quantifiers and connectives are verbal.
- Some conventional, mostly 2-place, predicates are symbolic, e.g. `=` and $<$, while most new predicates are verbal, e.g. *x is even.*

- Singular terms are often symbolic, e.g. `x + y`, but also often verbal, e.g. *the greatest prime factor of x.*
- A symbolic expression may not contain parts expressed in words.

The last rule has an important consequence for a symbolic predicate that is given a verbal argument: either the predicate is rendered verbally,

```
the greatest prime factor of x is equal to x/2
```

or a symbol is locally defined for the argument,

```
p = x/2, where p is the greatest prime factor of x.
```

If we want a proof assistant to generate text of publication quality, we have to obey these rules. They are also important in educational applications [8]. Correct verbalization is a part of standard mathematical language as much as correct symbolic notation is. The symbolic notation problem is to a large extent solved in computer algebras and many proof assistants, e.g. by using TeX. But verbalization is a more difficult problem, since it involves so much linguistics.

On the topmost level of definitions and proofs, the main problem is proper order and structure, which questions are studied in e.g. [9] and [33]. What a grammar library can contribute is the replacement of concrete text by abstract syntax trees. For instance, the formula *we have n cases* can be expressed by

```
mkCl (mkNP we_Pron) have_V2 (mkNP n case_N)
```

which will produce the correct forms for any language as function of n.

Going down to the level of propositional structure, we need to express quantifiers and connectives by using natural language. It is easy use the GF Resource Grammar Library to write rules such as

```
Conj A B = coord and_Conj A B   -- A and B
Disj A B = coord or_Conj A B    -- A or B
```

But now we encounter the problem of ambiguity. We cannot just compositionally translate `A & (B v C)` with *A and B or C.* This problem is solved in [6] by the use of grouping (indentation, bullets) and subsentential coordination (*x is prime and y is even or odd*), also known as **aggregation** in natural language generation literature. The translation of logical structures, although tricky, can be solved generically in the implementation of a proof system and then reused in applications on any domain, largely independently of language.

The vast majority of textual rendering problems is encountered when new concepts are defined by users as a part of proof construction. In [16], we presented a plug-in to the Alfa proof editor, where definitions can be seen as GF abstract syntax rules and annotated by concrete syntax rules telling how the new concepts are expressed in natural language. This idea is applicable in any proof assistant that is able to express the types of new concepts; it is also used in the KeY program verification system [2, 6].

At the time of [16], the GF Resource Grammar Library was not available, and users had to think of both grammar rules and mathematical idioms when writing annotations. For the WebALT project [8], a combinator library was written to cover the most frequent needs. It has a set of predication rules (`pred`) building clauses (`Cl`) to express prepositions, and a set of application rules (`app`) building noun phrases (`NP`) to express individual objects:

```
pred : V  -> NP -> Cl        -- x converges
pred : V  -> ListNP -> Cl    -- x, y and z intersect
pred : V2 -> NP -> NP -> Cl  -- x intersects y
pred : A  -> NP -> Cl        -- x is even
pred : A  -> ListNP -> Cl    -- x, y and z are equal
pred : A2 -> NP -> NP -> Cl  -- x is divisible by y
pred : N  -> NP -> Cl        -- x is a prime
pred : N  -> ListNP -> Cl    -- x, y and z are relative primes
pred : N2 -> NP -> NP -> Cl  -- x is a divisor of y

app  : N2 -> NP -> NP        -- the successor of x
app  : N2 -> ListNP -> NP    -- the sum of x, y and z
app  : N3 -> NP -> NP -> NP  -- the interval from x to y
```

Together with morphological paradigms, these combinators can be used for defining renderings of mathematical concepts easily and compactly.

The `pred` functions are very much like the `mkCl` constructors in Section 6.6 above, with the difference that the predicate (verb or adjective) is given the first argument position. In typical linearization rules for mathematical predicates, this has the advantage of permitting the use of partial application. This is what we do in the following examples, which produce English, French, and Finnish:

```
Succ : Nat -> Nat
Succ = app (mkN2 "successor")
Succ = app (mkN2 "successeur")
Succ = app (mkN2 "seuraaja")

Div : Nat -> Nat -> Prop
Div = pred (mkA2 "divisible" "by")
Div = pred (mkA2 "divisible" "par")
Div = pred (mkA2 "jaollinen" adessive)

Prime : Nat -> Prop
Prime = pred (mkA "prime")
Prime = pred (mkA "premier")
Prime = pred (mkA "jaoton")
```

8.1 A functor implementation of the translator

In the previous example, the languages use the syntax API in exactly the same way, and differ only in the lexicon. This results in repetition of code, which can actually be avoided in GF by using a **functor**. A functor in GF is, like in ML, a module that depends on **interfaces** (called **signatures** in ML), that is, modules that only show the types of constants and omit definitions. The resource API itself is an interface. In a typical GF application, the programmer builds herself another one to define a **domain lexicon**. In the present example, the domain lexicon interface declares the constants

```
successor_N2 : N2
divisible_A2 : A2
prime_A : A
```

The functor can then be written

```
Succ = app successor_N2
Div = pred divisible_A2
Prime = pred prime_A
```

To add a new language to a multilingual grammar implemented with a functor like this, it is enough to write a new instance of the domain lexicon. This technique is explained in more detail in [26] and [23].

9 Related work

9.1 Grammar formalisms

The fundamental idea of GF is the division of a grammar into an abstract and concrete syntax. It is this division that at the same time supports the communication between grammatical structures and other data structures, and permits the construction of multilingual grammars. The idea is first found in Curry [11], and it was used by Montague [21] with the purpose of giving denotational semantics to a fragment of English.

In linguistics, Curry's idea was for a long time ignored, with Montague as one of the few exceptions. The situation has however changed in the past ten years, with GF and related formalisms: ACG (Abstract Categorial Grammars [12]), HOG (Higher-Order Grammar, [24]), and Lambda Grammars [22]. Only GF has so far produced a large-scale implementation and grammar libraries.

9.2 Grammar libraries

CLE (Core Language Engine [28]) has been the closest point of comparison as for both the coverage and purpose of the GF Resource Grammar Library. The languages included in CLE are English, Swedish, French, and Danish, with similar but structurally distinct fragments covered. The "glue" between the languages is called QLF (Quasi-Logical Form), which in fact gives a structure rather similar to the ground API of the GF Resource Grammar. Like GF, CLE adresses the idea of sharing code between grammar implementations. It uses macros and file includes instead of functions and modules for this purpose.

Moreover, specializing a large grammar to a small application is adressed by a technique called explanation-based learning. The effect is often similar to GF's partial evaluation.

The LinGO Matrix project [3] defines a methodology for building grammars of different languages that cover the same phenomena. The language-independent syntax API of GF can be seen as an extreme version of the same idea, using a common formal representation for all languages. LinGO grammars are aimed to parse real texts.

Pargram [7] is a project with the goal of building a set of parallel grammars. Its original purpose was machine translation between English, French, and German. The grammars are connected with each other by transfer functions, rather than a common representation. Currently the project covers more than ten languages.

9.3 Compiling to natural language

When introducing a proof-to-text translator, the paper [10] opens a new perspective to the problem: natural language generation is seen as similar to compilation. Thus it uses code optimization techniques to improve the generated text, in a partly similar way to what in main-stream natural language generation is known as aggregation. These optimizations operate on what in a compiler would be called intermediate code—a level between source code (Coq proofs) and target code (natural language). Now, if we see natural language as target code, or machine code, we can understand the nature of some of the difficulties of generating it. Natural language, just like machine code, is difficult to deal with directly: it is better to hide it under a level of abstraction. Assembly code is one such level, but it is even better if the compiler can generate intermediate code retargetable to many machines. The intermediate code of [10] was indeed retargeted to English and French. The language-independent abstract syntax of GF Resource Grammar Library can be seen as a format implementing the same idea, now retargeted to more languages.

10 Conclusion

We have discussed the GF Resource Grammar Library from the point of view of library-based software engineering. This is a novel perspective on grammars, which are usually seen as programs performing parsing and generation, rather than as libraries. Libraries are useful for software localization and applications such as proof assistants. A main problem is to make the library API intelligible for non-linguist users. The use of a common API for different languages helps considerably; another useful instrument is overloading, which helps to keep the functions names memorizable and to create different views of the library.

Acknowledgments

It was Gilles Kahn who turned my attention to compiler techniques in natural language generation. I am deeply grateful for the invitation to visit him in Sophia-Antipolis in 1995, and also for the contact we had afterwards. GF was born from a combination of these compiler techniques and the type-theoretical view of abstract and concrete syntax, for which I am grateful to Per Martin-Löf.

The GF Resource Grammar Library has been built as a cooperation of many people, and gained from comments and suggestions from even more—to mention just some: Krasimir Angelov, Jean-Philippe Bernardy, Lars Borin, Björn Bringert, Lauri Carlson, Robin Cooper, Ali El Dada, Hans-Joachim Daniels, Elisabet Engdahl, Markus Forsberg, Harald Hammarström, Kristofer Johannisson, Janna Khegai, Peter Ljunglöf, Wanjiku Ng'ang'a, Bengt Nordström, and Jordi Saludes. During the past year, collaboration with the Software Methodology and Systems group at Chalmers (Andreas Priesnitz, Sibylle Schupp, and Marcin Zalewski) has brought to my attention the important conceptual work on software libraries done in the C++ community.

The research has been supported by the grant *Library-Based Grammar Engineering* from Vetenskapsrådet.

References

[1] A. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

[2] B. Beckert, R. Hähnle, and P. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS.* Springer-Verlag, 2006.

[3] Emily M. Bender and Dan Flickinger. 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, 2005.

[4] Bescherelle. *La conjugaison pour tous.* Hatier, 1997.

[5] B. Bringert. Embedded Grammars. MSc Thesis, Department of Computing Science, Chalmers University of Technology, 2004.

[6] D. A. Burke and K. Johannisson. 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 3402 of *LNCS/LNAI*, pages 51–66. Springer, 2005.

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

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

[9] Y. Coscoy. *Explication textuelle de preuves pour le calcul des constructions inductives*. PhD thesis, Université de Nice-Sophia-Antipolis, 2000.

[10] Y. Coscoy, G. Kahn, and L. Thery. Extracting text from proofs. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proc. Second Int. Conf. on Typed Lambda Calculi and Applications*, volume 902 of *LNCS*, pages 109–123, 1995.

[11] H. B. Curry. 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, 1963.

[12] Ph. de Groote. Towards Abstract Categorial Grammars. In *Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter, Toulouse, France*, pages 148–155, 2001.

[13] A. El Dada. Implementation of the Arabic Numerals and their Syntax in GF. In *Computational Approaches to Semitic Languages: Common Issues and Resources, ACL-2007 Workshop, June 28, 2007, Prague*, 2007.

[14] A. El Dada and A. Ranta. Implementing an Open Source Arabic Resource Grammar in GF. In M. A. Mughazy, editor, *Perspectives on Arabic Linguistics XX*, pages 209–232. John Benjamins, Amsterdam and Philadelphia, 2007.

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

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

[17] Gerard Huet. A Functional Toolkit for Morphological and Phonological Processing, Application to a Sanskrit Tagger. *The Journal of Functional Programming*, 15(4):573–614, 2005.

[18] M. Humayoun. Urdu Morphology, Orthography and Lexicon Extraction. MSc Thesis, Department of Computing Science, Chalmers University of Technology, 2006.

[19] J. Khegai. GF Parallel Resource Grammars and Russian. In *Coling/ACL 2006*, pages 475–482, 2006.

[20] Per Martin-Löf. Constructive mathematics and computer programming. In Cohen, Los, Pfeiffer, and Podewski, editors, *Logic, Methodology and Philosophy of Science VI*, pages 153–175. North-Holland, Amsterdam, 1982.

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

[22] 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.

[23] N. Perera and A. Ranta. 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*, 2007.

[24] C. Pollard. Higher-Order Categorial Grammar. In M. Moortgat, editor, *Proceedings of the Conference on Categorial Grammars (CG2004), Montpellier, France*, pages 340–361, 2004.

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

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

[27] A. Ranta. Grammatical Framework Homepage, 2008. `digitalgrammars.com/gf`.

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

[29] B. Stroustrup. *The C++ Programming Language, Third Edition.* Addison-Wesley, 1998.

[30] The Coq Development Team. The Coq Proof Assistant Reference Manual. `pauillac.inria.fr/coq/`, 1999.

[31] A. Trybulec. The Mizar Homepage. `http://mizar.org/`, 2006.

[32] M. Wenzel. Isar - a Generic Interpretative Approach to Readable Formal Proof Documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics, TPHOLs'99*, volume 1690 of *LNCS*, 1999.

[33] F. Wiedijk. Formal Proof Sketches. In *Types for Proofs and Programs*, LNCS 3085, pages 378–393. Springer, 2004.