GF and the Language of Mathematics

Aarne Ranta (aarne@chalmers.se)

Bonn, 18 December 2009

Plan

Introduction to GF

A grammar for mathematical text

I Introduction to GF

What is GF: Linguist’s view

A multilingual grammar formalism based on tectogrammatical representations

```
  Divisible         6 is divisible by 2
   /     \          6 è divisibile per 2
  /       \         6 ist teilbar durch 2
 6         2        6 on jaollinen 2:lla
```

What is GF: Mathematician's view

free algebras with reversible homomorphic mappings to strings

- *Math* = <*Exp*,*Prop*,*Div* : *Exp* -> *Exp* -> *Prop*>
- *Div*(*x*,*y*)* = concat(*x**, "is divisible by", *y**)
- *Div*(*x*,*y*)* = concat(*x**, "è divisible per", *y**)

What is GF: Programmer’s view

a special-purpose functional programming language for writing translators

```
fun Divisible : Exp -> Exp -> Prop

lin Divisible x y = x ++ "is divisible by" ++ y
lin Divisible x y = x ++ "è divisible per" ++ y
lin Divisible x y = x ! Nom ++ "ist teilbar durch" ++ y ! Acc
lin Divisible x y = x ! Nom ++ "on jaollinen" ++ y ! Adess
```

Background

Mathematics:

- Martin-Löf's type theory (1970)
- Logical frameworks (LF, ALF, Coq)

Linguistics:

- Curry's tectogrammar + phenogrammar (1961)
- Montague grammar (1970)

Functional programming: ML, Haskell, types, modules

Industry: Multilingual Document Authoring project (Xerox, 1998)

A first example

Abstract syntax (`cat`egory and `fun`ction declarations)

```
cat Exp
cat Prop
fun Even : Exp -> Prop
```

Concrete syntaxes (`lin`earization rules)

```
lin Even x = x ++ "is"  ++ "even"   -- English

lin Even x = x ++ "est" ++ "pair"   -- French

lin Even x = x ++ "ist" ++ "gerade" -- German
```

Linguistic motivation

Translation must preserve meaning

Abstract syntax serves as an **interlingua**

- hub of translation
- semantic structure expressed in type theory
- limitation to specific domain

Thus we use LF as a **framework for interlinguas**

Authoring help

Incremental translation: like T9, but grammar-aware.

Web demos: `tournesol.cs.chalmers.se:41296`

(Incremantal parsing of PMCFG: Angelov, EACL 2009)

Strings are not enough

The French and German rules don't scale up

**la somme de x et de y est pair*

la somme de x et de y est paire

**wenn x ist gerade, x+2 ist gerade*

wenn x gerade ist, ist x+2 gerade

Solution: parameters and linearization types

French:

```
param Gender = Masc | Fem ;

lincat Nat  = {s : Str ; g : Gender} ;
lincat Prop = {s : Str} ;

lin Even x = {
  s = x.s ++ "est" ++ case x.g of {
    Masc => "pair" ;
    Fem  => "paire"
    }
  } ;
```

German: parametrized word order

```
param Order = Main | Inverse | Subordinate ;

lincat Nat  = {s : Str} ;
lincat Prop = {s : Order => Str} ;

lin Even x = {
  s = \\o => case o of {
    Main        => x.s ++ "ist" ++ "gerade" ;
    Inverse     => "ist" ++ x.s ++ "gerade" ;
    Subordinate => x.s ++ "gerade" ++ "ist"
    }
  } ;
```

Too much code to write?

```
lin Even x = {
  s = x.s ++ "est" ++ case x.g of {
    Masc => "pair" ;
    Fem  => "paire"
    }
  } ;

lin Odd x = {
  s = x.s ++ "est" ++ case x.g of {
    Masc => "impair" ;
    Fem  => "impaire"
    }
  } ;
```

The functional programmer's solution

Introduce auxiliary functions (`oper`ations)

```
oper regA : Str -> Gender -> Str = \noir,g ->
  case g of {
    Masc => noir ;
    Fem  => noir + "e"
    } ;

lin Even x = {
  s = x.s ++ "est" ++ regA "pair" x.g
  } ;
lin Odd x = {
  s = x.s ++ "est" ++ regA "impair" x.g
  } ;
```

The advanced functional programmer's solution

Introduce higher-order functions

```
oper
  predA : (Gender -> Str) -> {s : Str ; g : Gender} -> {s : Str} = \bon,x -> {
    s = x.s ++ "est" ++ bon x.g
  } ;

lin Even = predA (regA "pair") ;
lin Odd = predA (regA "impair") ;
```

Resource grammar libraries

Operations can be stored in libraries, written by linguists.

Application programmers use **linguistic structures** in concrete syntax

```
lin Even x = mkCl x (mkA "even")
```

rather than strings:

```
lin Even x = x.s ++ "is" ++ "even"
```

Application programmers need not know low-level linguistic details

- parameters
- inflection
- word order

The GF resource grammar library

Core syntax + complete inflectional morphology + small lexicon.

Size: 70 categories, 180 functions, 150 kLOC, 5 person years, 20 programmers.

Languages: 16 finished (Bulgarian, Catalan, Danish, Dutch, English, Finnish, French, German, Interlingua, Italian, Norwegian, Polish, Romanian, Russian, Spanish, Swedish), 5 more partially available (Arabic, Hindi/Urdu, Latin, Thai, Turkish). ca. 12 more under construction.

Applications:

- software specifications (KeY project)
- mathematical exercises (WebALT project)
- dialogue systems (TALK project)
- multilingual translation on the web (MOLTO project)

Also: to show that GF scales up to wide-coverage grammars.

The organization of the resource grammar library

Language-independent syntax API `Syntax`

- all languages have S, NP, VP, etc, and same the rules for combining them

Language-dependent morphological API's `Paradigms`*Lang*

- languages differ in the complexity and variation of inflection

Naming conventions

`mk`*C*: syntactic operations for constructing objects of category *C*

word_C: words of category *C*

`mk` operations are **overloaded**:

```
  mkCl : NP -> V -> Cl             -- x diverges
  mkCl : NP -> V2 -> NP -> Cl      -- x intersects y
  mkCl : NP -> A -> Cl             -- x is even
  mkCl : NP -> A2 -> NP -> Cl      -- x is divisible by y
```

The module system of GF

Modules of different types:

```
abstract A      = {cat... fun...}

concrete C of A = open R in {lincat... lin...}

resource R      = {param... oper...}
```

The `resource` modules are eliminated from run-time grammars during compilation.

Opening a module

A module may **open** other modules:

```
  resource SyntaxEng = ...

  concrete MathEng of Math = open SyntaxEng in { ... }
```

The contents of the opened module are usable, but they are not inherited.

Name clashes are avoided by explicit qualification: `SyntaxEng.mkS`

Extending a module

A module of any type `module` can **extend** modules of the same type

```
abstract Logic = ...

abstract Arithmetic = Logic ** ...
abstract Geometry   = Logic ** ...

abstract Maths = Arithmetic, Geometry ** ...
```

Extending means **inheritance** of the contents of the module.

Using the resource grammar library

Here is one typical way to use the resource library:

```
abstract Math = {
  cat Exp ; Prop ;
  fun And : Prop -> Prop -> Prop ; Even : Exp -> Prop ;
}

concrete MathEng of Math = open SyntaxEng, ParadigmsEng in {
  lincat
    Prop = S ;
    Exp = NP ;
  lin
    And A B = mkS and_Conj A B ;
    Even x = mkS (mkCl x (mkA "even")) ;
}
```

The resource API is language-independent

Here is what we wrote for English:

```
concrete MathEng of Math = open SyntaxEng, ParadigmsEng in {
  lincat
    Prop = S ; Exp = NP ;
  lin
    And A B = mkS and_Conj A B ;
    Even x = mkS (mkCl x (mkA "even")) ;
}
```

In French, we can write

```
concrete MathFre of Math = open SyntaxFre, ParadigmsFre in {
  lincat
    Prop = S ; Exp = NP ;
  lin
    And A B = mkS and_Conj A B ;
    Even x = mkS (mkCl x (mkA "pair")) ;
}
```

Common syntax interface

Starting point of GF: semantic structures are language-independent.

Later observation: also syntactic structures are largely the same.

Advantages:

- comparative linguistics
- common API for programmers
- the possibility of parametrized implementations

Splitting a resource into an interface and its instance

Example: fragment of GF resource grammar library

```
interface Syntax = {           instance SyntaxEng of Syntax = {
oper                           oper
  S     : Type ;                 S     = ...
  Cl    : Type ;                 Cl    = ...
  NP    : Type ;                 NP    = ...
  V     : Type ;                 V     = ...
  A     : Type ;                 A     = ...
  mkS   : Cl -> S ;              mkS   = ...
  mkCl  : NP -> A -> Cl ;        mkCl  = ...
}                              }
```

Cf. signature and structure in ML.

Also: a genaralization of `abstract` vs. `concrete`.

Incomplete=parametrized module = functor

I.e. a module that opens an interface

```
incomplete concrete MathI of Math = open Syntax in {
  lincat Prop = S ; Exp = NP ;
  lin And A B = mkS and_Conj A B ;
}
```

We can avoid the repetition of code.

How to deal with *even* vs. *pair*? Write another interface!

Domain lexicon as interface and instances

Domain lexicon, a.k.a. terminology

```
interface LexMath =
  open Syntax in {
    oper
      even_A : A ;
      prime_A : A ;
}
```

```
instance LexMathEng of LexMath =
  open SyntaxEng, ParadigmsEng in {
   oper
     even_A = mkA "even" ;
     prime_A = mkA "prime" ;
```

Instantiating a functor

Provide instances to each opened interface: given

```
incomplete concrete MathI of Math = open Syntax, LexMath in ...
```

we can write

```
concrete MathEng of Math = MathI with
  (Syntax = SyntaxEng), (LexMath = LexMathEng) ;
```

and then also

```
concrete MathFre of Math = MathI with
  (Syntax = SyntaxFre), (LexMath = LexMathFre) ;
concrete MathGer of Math = MathI with
  (Syntax = SyntaxGer), (LexMath = LexMathGer) ;
```

The modules in a typical application

COMMON: Abstract syntax

```
abstract Math = {...}
```

COMMON: Domain lexicon interface

```
interface LexMath = open Syntax in {...}
```

COMMON: Top-level functor parametrized on `Syntax` and domain lexicon

```
incomplete concrete MathI of Math = open Syntax, LexMath in {...}
```

SPECIFIC: Domain lexicon instance

```
instance LexMathEng of LexMath = open SyntaxEng, ParadigmsEng in {...}
```

SPECIFIC: Top-level functor instantiation

```
concrete MathEng of Math = MathI with
  (Syntax = SyntaxEng), (LexMath = LexMathEng) ;
```

Porting the application to a new language

Write an instance of the lexicon interface

```
instance LexMathFin of LexMath = open SyntaxFin, ParadigmsFin in {
  oper
    even_A = mkA "parillinen" ;
    prime_A = mkA "jaoton" ;
}
```

Mechanically provide an instantiation of the top-level functor

```
concrete MathFin of Math = MathI with
  (Syntax = SyntaxFin),
  (LexMath = LexMathFin) ;
```

Why this works

Logical structures are expressed with the same syntactic structures in different languages...

...even though `Syntax` is implemented differently in different languages

But, of course, words are different in different languages - hence the `Lex` interface (usually domain-specific).

Discrepancies in the use of the functor

Sometimes the semantics is not expressed by the same syntactic structure.

English: *x is prime* (an adjective)

Finnish: *x on alkuluku* (a noun: "x is a prime-number")

Possible solution: make the functor and the lexicon interface more general

```
  lin Even x = mkS x prime_VP ;
```

```
  oper prime_VP : VP ;
```

But this is not stable when new languages are added.

Solving discrepancies by restricted inheritance

```
concrete MathFin of Math = MathI - [Prime] with
  (Syntax = SyntaxFin),
  (LexMath = LexMathFin) ** open ParadigmsFin in {

  lin Prime x = mkS x (mkVP (indefNP (mkN "alkuluku"))) ;
  }
```

Adding precision via dependent types

```
cat
  Prop ;
  Dom ;
  Elem (x : Dom) ;
fun
  Set, Number : Dom ;
  Empty : Elem Set -> Prop ;
  Even : Elem Number -> Prop ;
  Equal : (D : Dom) -> Elem D -> Elem D -> Prop ;
  EmptySet : Elem Set ;
  Forall : (A : Dom) -> (Elem A -> Prop) -> Prop ;
```

Proof checking as type checking

```
cat
  Text ;
  Proof (P : Prop) ;
fun
  EmptyIsEmpty : Proof (Empty EmptySet) ;

  ReflEq : (A : Dom) -> (a : Elem A) -> Proof (Equal A a a) ;

  UnivI : (A : Dom) -> (B : Elem A -> Prop) ->
    ((x : Elem A) -> Proof (B x)) -> Proof (Forall A B) ;
```

Obtaining GF

Homepage `grammaticalframework.org`

Binaries for Linux, Mac OS, Windows

Open-source software

Latest versions with Darcs version controlled (mirrored in read-only Subversion)

On-line documentation

II A grammar for mathematical text

Choices

Unrestricted language vs. controlled language

Full semantic controlled vs. surface syntax

Text structure vs. sentences only

User-extensible vs. static

Choices

Unrestricted language vs. **controlled language**

Full semantic controlled vs. **surface syntax**

Text structure vs. sentences only

User-extensible vs. static

Goal

Controlled language for mathematical proofs

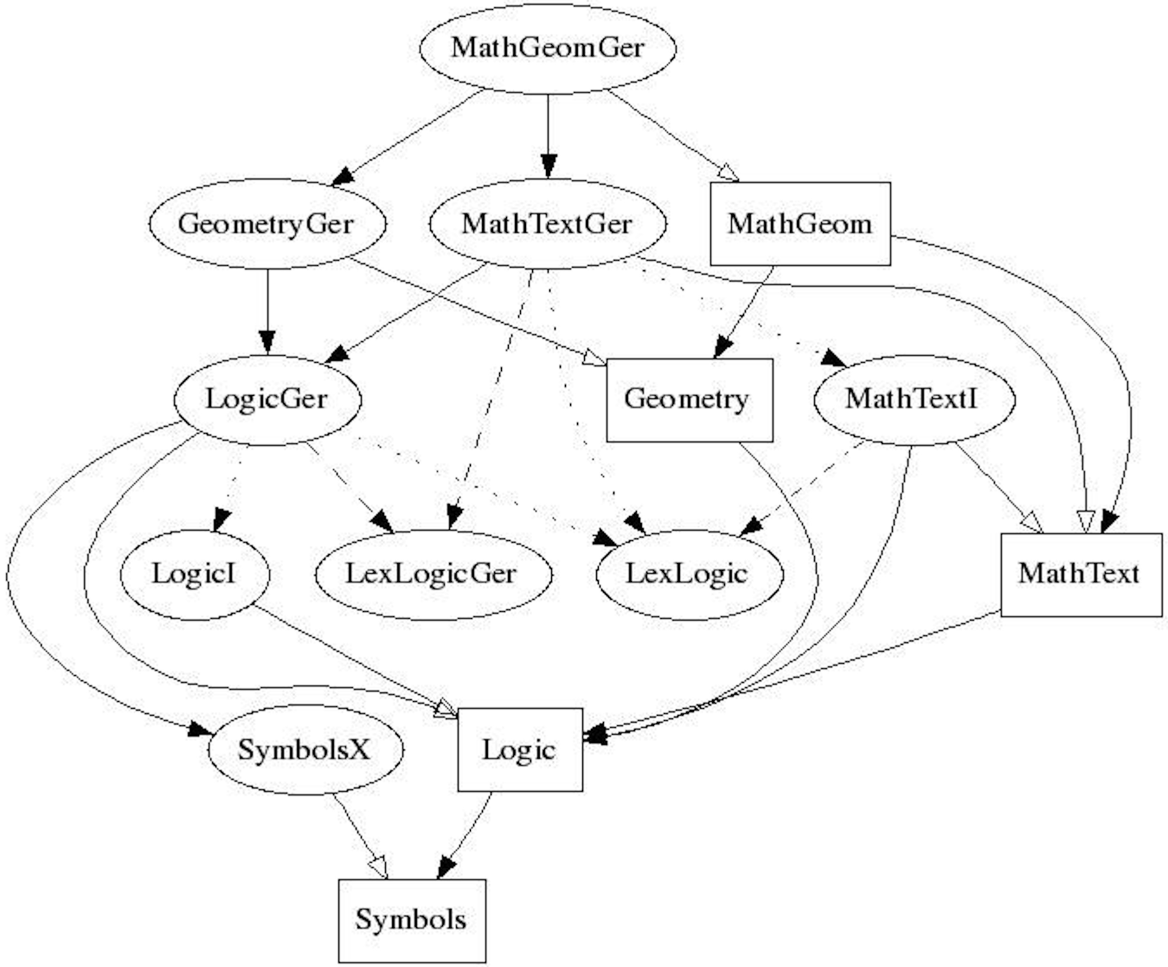
Mixture of English/French/German and LaTeX

Coverage of Naproche and beyod

Plugin to proof systems with

- incremental parsing
- generation, translation

Implementation


MathGeomGer
GeometryGer
MathTextGer
MathGeom
LogicGer
Geometry
MathTextI
LogicI
LexLogicGer
LexLogic
MathText
SymbolsX
Logic
Symbols

Implementation

```
 12 Geometry.gf     18 GeometryEng.gf  18 GeometryFre.gf  19 GeometryGer.gf
 20 LexLogic.gf     15 LexLogicEng.gf  16 LexLogicFre.gf  16 LexLogicGer.gf
 24 Logic.gf         5 LogicEng.gf     15 LogicFre.gf     15 LogicGer.gf
              48 LogicI.gf
  1 MathGeom.gf      4 MathGeomEng.gf   4 MathGeomFre.gf   4 MathGeomGer.gf
 45 MathText.gf      5 MathTextEng.gf   5 MathTextFre.gf   5 MathTextGer.gf
              74 MathTextI.gf
 10 Symbols.gf
              15 SymbolsX.gf
413 total
```

Grammar

Let's

- look at the source code
- try it out
- add some concepts

The source code is available in `GF/examples/mathtext`.