Parametrized Modules in Grammar Engineering

Aarne Ranta

Colloquium in Honor of Gérard Huet, Paris 22-23 June 2007

Plan

Grammar formalisms as programming languages

Overview of GF

The module system of GF

Parametrized modules

Linguistic result: interlingua vs. transfer

Linguistic result: language families

I Grammar formalisms as programming languages

Grammar Formalisms

Languages for defining grammars

Used by compiler writers: BNF, EBNF, YACC,...

Used by linguists: DCG, PATR, HPSG, LFG, TAG, CCG, GSL, XFST, IG, Regulus, ACG, HOG, GF,...

What is a formalism? A double view

Low level

- machine language / mathematical model
- austere, non-redundant
- easy to implement / reason about

High level

- tool for programming
- rich, redundant
- easy to program in
- redundancy gives safety (e.g. static type checking)

How grammar formalisms are viewed

Focus on low level, because of

- need to settle questions of complexity and expressivity
- heritage of old languages (Lisp and Prolog)
- few users, who are experts

Static checking is rare

Generalizations via macros and file includes

At the same time, linguists love abstractions and generalizations!

Growing demand for grammars

Areas:

- information retrieval
- software localization
- speech-based interfaces

Non-linguist programmers need to write grammars

This requires

- high-level languages
- static checking
- libraries

From low to high level of language

Starting point: machine code

- repetitive code
- copy and paste

From low to high level of language

Starting point: machine code

- repetitive code
- copy and paste

First step: metalevel programming

- automatize programmer's operations on the code
- syntactic manipulation

From low to high level of language

Starting point: machine code

- repetitive code
- copy and paste

First step: metalevel programming

- automatize programmer's operations on the code
- syntactic manipulation

Second step: internalize the metalevel

- lift high-level concepts to the language
- give them semantics

Some internalizations

From **macros** to **functions**

From **file inclusions** to a **module system**

From **preprocessing** to **compilation**

- from syntactic manipulation to semantic analysis
- errors are captured at the level of source code
- libraries can be precompiled (and delivered in binary...)

Where do **parametrized modules** arise?

A view on parametrized modules

They internalize the configurable inclusion of macro files

Example: language-neutral Latex

```
% file drink.tex
\include{english}  %% \include{french}
\Qualify{\red}{\wine}

% file english.tex
\newcommand{\wine}{wine}
\newcommand{\red}{red}
\newcommand{\Qualify}[2]{#1 #2}

% file french.tex
\newcommand{\wine}{vin}
\newcommand{\red}{rouge}
\newcommand{\Qualify}[2]{#2 #1}
```

The "interface" is the macro names with their arities.

This idea is used in Regulus (Rayner & al.) to write multilingual grammars.

Functional programming in linguistics: Zen

The Zen morphology toolkit (Huet)

- library in OCaml
- static type checking, module system
- generation of several formats
- efficient and reliable production of language resources (e.g. Sanskrit)

Functional programming in linguistics: GF

GF = Grammatical Framework

- a language of its own
- high-level source language + simple "machine language" (Canonical GF)
- interpreters for Canonical GF: in Java, Haskell, C++, Prolog
- compilers from Canonical GF: to C, Javascript, GSL/Nuance, SRGS, HTK/ATK (speech recognizer language models)

From Zen: datastructures and algorithms:

- tries
- zippers

II Overview of GF

Background

GF = Logical Framework + concrete syntax

Tradition:

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

The "Curry architecture" has gained ground in the 2000's

- ACG (de Groote)
- HOG (Pollard)
- Lambda grammars (Muskens)

A first example

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

```
cat Nat
cat Prop
fun Even : Nat -> 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**

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 = predA (regA "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, 130 kLOC, 4 person years, 14 programmers.

Languages: 10 finished (Danish, English, Finnish, French, German, Italian, Norwegian, Russian, Spanish, Swedish), 5 under construction (Arabic, Catalan, Swahili, Thai, Urdu).

Applications:

- software specifications (KeY project)
- mathematical exercises (WebALT project)
- dialogue systems (TALK project)

Also: to show that GF scales up to large grammars.

The organization of the resource grammar library

Language-independent Syntax API

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

Language-dependent morphological Paradigms API's

- languages differ in the complexity and variation of inflection

Common syntax interface

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

Later observation: also syntactic structures are largely the same.

Advantages:

- comparative linguistics (cf. LinGO Matrix in HPSG)
- common API for programmers
- the possibility of parametrized implementations

III The module system of GF

The computation model of GF

Abstract syntax: LF

- free algebra of trees
- dependently typed, second-order function types (for HOAS)
- **syntax trees**: eta-long well-typed lambda terms

Concrete syntax

- homomorphism from trees to concrete syntax objects
- **concrete syntax objects**: nested tuples of strings and integers

With some restrictions on abstract syntax, the formalism is mildly context-sensitive, with polynomial parsing complexity (Ljunglöf 2004)

GF as a programming language

Dependently typed functional language with extra constructs

- finite functions (inflection tables)
- regular expression pattern matching

Module system inspired by Java, C++ and ML

- inheritance
- parametrized modules
- seven meanings of `include`

Overloading *à la* Ada, C++

The module system of GF

At run-time: one abstract + many concrete syntaxes

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

concrete C of A = {lin...}
```

In the source language, and at compile time: many 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, by inlining.

But they do have some separate compilation: type checking and partial evaluation.

Compilation example

```
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"
    }
  } ;
```

```
-- type Gender = Ints 2

-- lincat Nat  = [Ints 2, Str]
-- lincat Prop = [Str]

lin Even = [
  $0.1 ++ "est" ++ [
     "pair",
     "paire"
     ].($0.0)
  ]
```

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.

Changing an inherited module

The contents of an inherited module may not be changed.

But there is the possibility of **restricted inheritance**:

```
abstract IntLogic = Logic - [ExclMid, RAA] ** {
  fun RAA : ...
}
```

Diamond property: a multiply inherited name must come from a common base module.

Opening a module

A module of any type may **open** modules of any type

```
resource SyntaxEng = ...

concrete LogicEng of Logic = 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`

Splitting a resource into an interface and its instance

Example: fragment of GF resource grammar library

```
interface Syntax = {
oper
  S     : Type ;
  NP    : Type ;
  VP    : Type ;
  A     : Type ;
  mkS   : NP -> VP -> S ;
  mkVP  : A -> VP ;
  conjS : S -> S -> S ;
}
```

```
instance SyntaxEng of Syntax = {
oper
  S     = ...
  NP    = ...
  VP    = ...
  A     = ...
  mkS   = ...
  mkVP  = ...
  conjS = ...
}
```

Cf. signature and structure in ML.

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

Using the resource grammar library

Here is one way to use the resource library:

```
abstract Logic = {
  cat Prop ;
  fun And : Prop -> Prop ;
}
concrete LogicEng of Logic = open SyntaxEng in {
  lincat Prop = S ;
  lin And A B = conjS A B ;
}
```

What about French: do we have to write

```
concrete LogicFre of Logic = open SyntaxFre in {
  lincat Prop = S ;
  lin And A B = conjS A B ;
}
```

IV Parametrized modules

Incomplete=parametrized module = functor

Opening an interface is what makes a module parametrized:

```
incomplete concrete LogicI of Arithm = open Syntax in {
  lincat Prop = S ;
  lin And A B = conjS A B ;
}
```

The sense of this:

- logical structures are expressed with the same syntactic structures in different languages...
- ...even though `Syntax` is implemented differently in different languages

Instantiating a functor

Provide instances to each opened interface: given

```
incomplete concrete LogicI of Logic = open Syntax in ...
```

we can write

```
concrete LogicEng of Logic = LogicI with (Syntax = SyntaxEng) ;
```

and then also

```
concrete LogicFre of Logic = LogicI with (Syntax = SyntaxFre) ;
concrete LogicGer of Logic = LogicI with (Syntax = SyntaxGer) ;
concrete LogicIta of Logic = LogicI with (Syntax = SyntaxIta) ;
```

The modules in a typical application

Abstract syntax, possibly extending some base modules

```
abstract Arithm = Logic ** {
  cat Nat ;
  fun Even, Prime : Nat -> Prop ;
}
```

Domain-dependent lexicon interface

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

Top-level functor parametrized on resource grammar `Syntax` and domain lexicon

```
incomplete concrete ArithmI of Arithm = LogicI ** open Syntax, ArithmLex in {
  lincat Nat = NP ;
  lin Even x = mkS x (mkVP even_A) ;
  lin Prime x = mkS x (mkVP prime_A) ;
}
```

Porting the application to a new language

Write an instance of the lexicon interface

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

Mechanically provide an instantiation of the top-level functor

```
concrete ArithmFin of Arithm = LogicFin ** ArithmI with
  (Syntax = SyntaxFin),
  (ArithmLex = ArithmLexFin) ;
```

Discrepancies in the use of the functor

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

English: *x is prime* (an adjective)

Swedish: *x är ett primtal* (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 ArithmSwe of Arithm = LogicSwe ** ArithmI - [Prime] with
  (Syntax = SyntaxSwe),
  (ArithmLex = ArithmLexSwe) ** open ParadigmsSwe in {

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

Module system summary: seven meanings of ”include”

```
B = A ** ...                 -- inheritance
C = open R in ...            -- opening
concrete C of A = ...        -- concrete of abstact
instance J of I = ...        -- instance of interface
M = F with ...               -- instantiation of functor
M = ... with (I = ...)       -- interface in functor instantiation
M = ... with (... = J)       -- instance of interface in functor instantiation
```

V Linguistic results: interlingua vs. transfer

Interlingua vs. transfer, 1

Two alternative models of machine translation.

The basic translation model in GF is **interlingua**:

```
     två är ett primtal
            |                    parsing
         Prime two
            |                    linearization
       two is prime
```

Due to reversibility, a system with n languages needs n concrete syntax modules.

Interlingua vs. transfer, 2

**Transfer*: change the structure between source and target language

```
        två är ett primtal
                |                       parsing
  mkS två_PN (mkVP (indefNP primtal_N))
                |                       transfer
     mkS two_PN (mkVP prime_A)
                |                       linearization
           two is prime
```

A system with *n* languages needs *n*(*n*-1) transfer functions.

There is, moreover, runtime overhead.

Compile-time transfer

Transfer is needed when languages use different structures for the same thing.

In GF, this means replacing a functor-based concrete syntax rule.

The transfer is eliminated when the grammar is compiled.

```
    två är ett primtal                         två är ett primtal
           |                                           |
 mkS två_PN (mkVP (indefNP primtal_N))                 |
           |                                           |
       Prime two                                   Prime two
           |                                           |
 mkS two_PN (mkVP prime_A)                             |
           |                                           |
      two is prime                                two is prime
```

NB: some transfer cannot be eliminated at compile time.

VI Linguistic results: language families

Scandinavian and Romance

Parametrized modules are also used inside the resource grammar library

Shared functor code with language-specific instances for

- parameters and linearization types
- syntactic combination rules

(no effort to share morphology and lexicon code)

Two families are treated in this way:

- Scandinavian (Danish, Norwegian, Swedish), over 90% percent is shared
- Romance (French, Italian, Spanish), over 80% is shared.

Adding Catalan to the Romance family (Jordi Saludes, UPC) did not require changes in the interfaces and the functor.

Example shared rule: adjectival modification

The adjective agrees to the noun in gender: *nombre pair*, *somme paire*.

Both receive their number from an outer determiner: *chaque nombre pair*, *plusieurs nombres pairs*.

Their order depends on the adjective: *bon livre*, *livre ennuyeux*

```
lincat AP = {s : Gender => Number => Str ; isPre : Bool} ;

lincat CN = {s : Number => Str ; g : Gender} ;

lin AdjCN ap cn =
    let
      g = cn.g
    in {
      s = \\n => preOrPost ap.isPre (ap.s ! g ! n) (cn.s ! n) ;
      g = g ;
      } ;
```

The main differences

Which prepositions fuse with the definite article

- Fre and Spa *à, de*, Ita also *da, in, con, su*

Which auxiliary verbs are used in compound tenses

- Fre and Ita *avere* and *essere*, Spa only *haber*

Derivatively, if the participle can agree to the subject

- Fre *elle est partie*, Spa *ella ha partido*

If the participle agrees to the foregoing clitic

- Fre *il les a vues*, Spa *el las ha visto*

How infinitives and clitics are placed relative to each other

- Fre *la voir*, Ita *vederla*

How negative imperatives are formed

- Fre *ne me quitte pas*, Ita *non lasciarmi*

Whether a preposition is repeated in conjunction

- Fre *la somme de 3 et de 4*, Ita *la somma di 3 e 4*

The interface DiffRomance

```
Prepos        : Type ;
VType         : Type ;
partAgr       : VType -> VPAgr ;
vpAgrClit     : Agr -> VPAgr ;
clitInf       : Str -> Str -> Str ;
mkImperative : VP -> {s : Polarity => AAgr => Str} ;
conjunctCase : NPForm -> NPForm ;
```

Results for language families

Gives an answer to "how much of the grammar is really the same" in related languages

Functors save time in creation and maintenance

- most differences were identified in the beginning
- thus most extensions of the grammar were shared

Sometimes the functor is more complicated than a non-functor would be

- need to think about many languages at the same time
- but the third language required only little changes, the fourth required none

Restricted inheritance was not available, but should be useful

- often just one language of the four is deviant

Conclusions on the module system

Large parts of languages can share abstract syntax.

This enables functor-based use of libraries.

Adding a new language to a system is often just a matter of writing a lexicon instance.

Writing grammars only requires

- knowledge of domain vocabulary
- the applicational fragment of GF