

# Introduction to the Grammatical Framework

Iavor S. Diatchki

Galois Inc

Tech Talk

# This Talk

- 1 Introduction
- 2 Parsing C Types: An Extended Example
- 3 Using GF With Other Languages
- 4 Expressing Semantic Properties
- 5 Using Libraries
- 6 Coda

# The Grammatical Framework

- A special purpose language for grammars.
- Based on functional programming and logical frameworks.
- Good for language processing.
- First created in 1998, still actively developed.
- Current developers:
  - Krasimir Angelov,
  - Björn Bringert,
  - Aarne Ranta.
- Open source: `gf` tool GPL, libraries LGPL.

# Abstract Syntax (Semantics)

```
abstract Pred = {
```

```
}
```

A module defining an abstract syntax.

# Abstract Syntax (Semantics)

```
abstract Pred = {  
  cat  
    Noun; Verb; Sentence;  
  
}
```

Type declarations.

# Abstract Syntax (Semantics)

```
abstract Pred = {  
  cat  
    Noun; Verb; Sentence;  
  
  fun  
    Fact      : Noun -> Verb -> Noun -> Sentence;  
    John, Mary : Noun;  
    Love, Know : Verb;  
  
}
```

Constructor declarations.

# Abstract Syntax (Semantics)

```
abstract Pred = {  
  cat  
    Noun; Verb; Sentence;  
  
  fun  
    Fact      : Noun -> Verb -> Noun -> Sentence;  
    John, Mary : Noun;  
    Love, Know : Verb;  
  
  flags startcat = Sentence;  
}
```

Default type to parse/show.

# Concrete Syntax (Notation)

```
concrete PredEng of Pred = {
```

```
}
```

A module defining a concrete syntax.

# Concrete Syntax (Notation)

```
concrete PredEng of Pred = {
```

```
  lincat
```

```
    Noun, Verb, Sentence = Str;
```

```
-- Str is a built-in type for lists of tokens.
```

```
}
```

Concrete types to represent semantic types.

# Concrete Syntax (Notation)

```

concrete PredEng of Pred = {
  lincat
    Noun, Verb, Sentence = Str;

  lin
    John           = "John";
    Mary           = "Mary";
    Love           = "loves";
    Know           = "knows";
    Fact x f y     = x ++ f ++ y;
}

```

Concrete representations of constructors.

# GF Interpreter Commands

```
import PredEng.gf
```

# GF Interpreter Commands

```
import PredEng.gf  
  
parse "John loves Mary"  
> Fact John Love Mary
```

# GF Interpreter Commands

```
import PredEng.gf
```

```
parse "John loves Mary"
```

```
> Fact John Love Mary
```

```
linearize Fact John Know Mary
```

```
> John knows Mary
```

# GF Interpreter Commands

```
import PredEng.gf
```

```
parse "John loves Mary"
```

```
> Fact John Love Mary
```

```
linearize Fact John Know Mary
```

```
> John knows Mary
```

```
generate_random -number=3 | linearize
```

```
> John loves Mary
```

```
John knows Mary
```

```
John knows John
```

# A Different Concrete Syntax

```
concrete PredFre of Pred =  
lincat  
  Noun, Verb, Sentence = Str;  
  
lin  
  John           = "John";  
  Mary           = "Mary";  
  Love           = "aime";  
  Know           = "connue";  
  Fact x f y     = x ++ f ++ y;
```

# Translation

```
import PredEng.gf PredFre.gf
```

```
p -lang=PredEng "John loves Mary" | l -lang=PredFre  
> John aime Mary
```

parse may be abbreviated to p

linearize may be abbreviated to l

# Translation

```
import PredEng.gf PredFre.gf
```

```
p -lang=PredEng "John loves Mary" | l -lang=PredFre  
> John aime Mary
```

```
p -lang=PredFre "John aime Mary" | l -lang=PredEng  
> John loves Mary
```

parse may be abbreviated to p

linearize may be abbreviated to l

# Summary of Basic Concepts

- **abstract** modules define the *semantic concepts*.
- **concrete** modules define *syntactic notation* for concepts.
- **parse** identifies the meaning of text in a given notation.
- **linearize** expresses a concept in given notation.
- To translate we compose parsing and linearization.

- 1 Introduction
- 2 Parsing C Types: An Extended Example**
- 3 Using GF With Other Languages
- 4 Expressing Semantic Properties
- 5 Using Libraries
- 6 Coda

# The Abstract Syntax

```
abstract CType = {  
  flags startcat = Decl;  
  
  cat Decl;  
  fun hasType : String -> CType -> Decl;
```

# The Abstract Syntax

```
abstract CType = {  
  flags startcat = Decl;  
  
  cat Decl;  
  fun hasType : String -> CType -> Decl;  
  
  cat CType;  
  fun  
    float, double, void : CType;  
    ptr : CType -> CType;  
    funT : CType -> CType;  -- simpl: no arguments
```

# The Abstract Syntax

```
cat Sign; Integral;
```

```
fun
```

```
  integral : Sign -> Integral -> CType;
```

```
  signed, unsigned : Sign;
```

```
  char, short, int, long : Integral;
```

# The Abstract Syntax

```
cat Sign; Integral;  
fun  
  integral : Sign -> Integral -> CType;  
  signed, unsigned : Sign;  
  char, short, int, long : Integral;  
  
cat Dim;  
fun  
  arr      : Dim -> CType -> CType;  
  noSize   : Dim;  
  ofSize   : Int -> Dim;  
}
```

# Example of a Semantic Value

The declaration:

```
char *x[10]
```

is represented as:

```
hasType "x" (  
  arr (ofSize 10)  
    (ptr (integral signed char))  
)
```

# C Notation: Integral Types

```
concrete CTypeC of CType = {
```

```
  lincat
```

```
    Sign, Integral = Str;
```

```
  lin
```

```
    char          = "char";
```

```
    short         = "short" ++ (" " | "int");
```

```
    int           = "int" | "";
```

```
    long          = "long" ++ (" " | "int");
```

```
    signed        = "" | "signed";
```

```
    unsigned      = "unsigned";
```

# C Notation: Integral Types

```
concrete CTypeC of CType = {
```

```
  lincat
```

```
    Sign, Integral = Str;
```

```
  lin
```

```
    char          = "char";
```

```
    short         = "short" ++ (" " | "int");
```

```
    int           = "int" | "";
```

```
    long          = "long" ++ (" " | "int");
```

```
    signed        = "" | "signed";
```

```
    unsigned      = "unsigned";
```

Variants allow for alternative representations.

# Avoiding Repetition

**oper**

```
opt      : Str -> Str  
         = \s -> "" | s;
```

# Avoiding Repetition

**oper**

```
opt      : Str -> Str      -- Type  
         = \s -> "" | s;
```

# Avoiding Repetition

**oper**

```
opt      : Str -> Str  
= \s -> "" | s; -- Definition
```

# Avoiding Repetition

## oper

```
opt      : Str -> Str  
         = \s -> "" | s;
```

## lin

```
char     = "char";  
short    = "short" ++ opt "int";  
int      = "int" | "";  
long     = "long" ++ opt "int";  
signed   = opt "signed";  
unsigned = "unsigned";
```

# Avoiding Repetition

## oper

```
opt      : Str -> Str
         = \s -> "\"" | s;
```

## lin

```
char      = "char";
short     = "short" ++ opt "int";
int       = "int" | "";
long      = "long" ++ opt "int";
signed    = opt "signed";
unsigned  = "unsigned";
```

Leave as is, because we prefer `int` to be displayed.

# Representing Types

## **lincat**

```
CType      = TypeRep;
```

## **oper**

```
TypeRep   : Type  
           = { base, before, after : Str;  
              prec : Prec };
```

To represent C types we use a record type.

# Representing Types

## **lincat**

```
CType      = TypeRep;
```

## **oper**

```
TypeRep   : Type  
           = { base, before, after : Str;  
              prec : Prec };
```

```
-- char *x[10]
```

Field `base` is the basic part of the type.

# Representing Types

## **lincat**

```
CType      = TypeRep;
```

## **oper**

```
TypeRep   : Type  
           = { base, before, after : Str;  
              prec : Prec };
```

```
-- char *x[10]
```

Field `before` is the part of type before variable.

# Representing Types

## **lincat**

```
CType      = TypeRep;
```

## **oper**

```
TypeRep   : Type  
           = { base, before, after : Str;  
             prec : Prec };
```

```
-- char *x[10]
```

Field `after` is the part of type written after the variable.

# Representing Types

## **lincat**

```
CType      = TypeRep;
```

## **oper**

```
TypeRep   : Type  
           = { base, before, after : Str;  
              prec : Prec };
```

```
-- char *x[10]
```

## **param**

```
Prec      = Low | High;
```

Prec is a type used only in the concrete syntax.

# Notation for Basic Types

## oper

```
base      : Str -> TypeRep
= \s -> { base = s;
         before, after = "";
         prec = Low
       };
```

# Notation for Basic Types

## oper

```

base      : Str -> TypeRep
          = \s -> { base = s;
                  before, after = "";
                  prec = Low
                };

```

## lin

```

integral s t = base (s ++ t);
void          = base "void";
float        = base "float";
double       = base "double";

```

# Notation for Arrays and Functions

## oper

```
postfix: Str -> TypeRep -> TypeRep
  = \s,t ->
    { base    = t.base;
      before  = t.before;
      after   = s ++ t.after;
      prec    = High
    };
```

# Notation for Arrays and Functions

## oper

```

postfix: Str -> TypeRep -> TypeRep
  = \s,t ->
    { base    = t.base;
      before  = t.before;
      after   = s ++ t.after;
      prec    = High
    };

```

**lincat** Dim = Str;

## lin

```

noSize    = "[" ++ "]";
ofSize n  = "[" ++ n.s ++ "]";

```

# Notation for Arrays and Functions

## oper

```

postfix: Str -> TypeRep -> TypeRep
  = \s,t ->
    { base    = t.base;
      before  = t.before;
      after   = s ++ t.after;
      prec    = High
    };

```

**lincat** Dim = Str;

## lin

```

noSize    = "[" ++ "]";
ofSize n  = "[" ++ n.s ++ "]";
arr       = postfix;
funT      = postfix ( "(" ++ ")" ); -- no args.

```

# Notation for Pointers

## oper

```
mkPtr: Str -> Str -> TypeRep -> TypeRep
  = \l,r,t ->
    { base    = t.base;
      before  = t.before ++ l ++ "*";
      after   = r ++ t.after;
      prec    = Low
    };
```

```
-- char (*p)[20]
```

# Notation for Pointers

## oper

```
mkPtr: Str -> Str -> TypeRep -> TypeRep
  = \l,r,t ->
    { base    = t.base;
      before  = t.before ++ l ++ "*";
      after   = r ++ t.after;
      prec    = Low
    };
```

```
-- char (*p)[20]
```

## lin

```
ptr t    = case t.prec of {
           Low  => mkPtr "" "" t;
           High => mkPtr "(" ")" t
         };
```

# Notation for Declarations

```
lincat Decl = Str;  
lin  
  hasType x t = t.base ++ t.before ++ x.s ++ t.after;  
}
```

# Notation for Declarations

```

lincat Decl = Str;
lin
  hasType x t = t.base ++ t.before ++ x.s ++ t.after;
}

import CTypeC.gf
ps -lexcode "char (*p[10])[20]" | parse

> hasType "p"
  (arr (ofSize 10)
    (ptr (arr (ofSize 20)
              (integral signed char))))

```

- 1 Introduction
- 2 Parsing C Types: An Extended Example
- 3 Using GF With Other Languages**
- 4 Expressing Semantic Properties
- 5 Using Libraries
- 6 Coda

# Compiled GF

- PGF: Portable Grammar Format
- Compiled representation of concrete and abstract syntaxes.
- Generated like this:

```
> gf --make *.gf
> ls *.pgf
CType.pgf
```
- By default, named after abstract syntax.

# A Haskell API for PGF

**module PGF where**

`readPGF :: FilePath -> IO PGF`

`startCat :: PGF -> Type`

`languages :: PGF -> [Language]`

`parse :: PGF -> Language -> Type -> String -> [Tree]`

`linearize :: PGF -> Language -> Tree -> String`

...

# Example: Translator

```
main = do pgf <- readPGF "CType.pgf"  
         txt <- getTxt  
         c   <- getLang pgf "CTypeC"  
         eng <- getLang pgf "CTypeEng2"  
         case parse pgf c (startCat pgf) txt of  
           e : _ -> putStrLn (linearize pgf eng e)  
           _     -> die ("Cannot parse: " ++ txt)
```

# Example: Translator

```

main = do pgf <- readPGF "CType.pgf"
         txt <- getTxt
         c   <- getLang pgf "CTypeC"
         eng <- getLang pgf "CTypeEng2"
         case parse pgf c (startCat pgf) txt of
           e : _ -> putStrLn (linearize pgf eng e)
           _     -> die ("Cannot parse: " ++ txt)

getLang pgf name =
  case [ l | l <- languages pgf,
         name == showLanguage l ] of
    lang : _ -> return lang
    []       -> die ("Cannot find language " ++ name)

```

# Working With the AST in Haskell

- From a PGF file, GF can generate:
  - Haskell datatypes for abstract syntax
  - Coersion functions to and from the Tree type.

```
> gf --batch --output-format=haskell CType.pgf
> ls *.hs
CType.hs
```

- GF supports other output formats too (e.g., JavaScript).

# Generated Code

```
class Gf a where
  gf :: a -> Tree
  fg :: Tree -> a

data GCType =
  Garr GDim GCType
| Gdouble
| Gfloat
| GfunT GCType
| Gintegral GSign GIntegral
...
deriving Show

instance Gf GCType where ...
```

# Example: Dump AST

```
main = do pgf <- readPGF "CType.pgf"  
      txt <- getTxt  
      c   <- getLang pgf "CTypeC"  
      case parse pgf c (startCat pgf) txt of  
        e : _ -> putStrLn (ppShow (fg e :: GDecl))  
        _     -> die ("Cannot parse: " ++ txt)
```

## Example: Dump AST

```

main = do pgf <- readPGF "CType.pgf"
      txt <- getTxt
      c   <- getLang pgf "CTypeC"
      case parse pgf c (startCat pgf) txt of
        e : _ -> putStrLn (ppShow (fg e :: GDecl))
        _   -> die ("Cannot parse: " ++ txt)

```

```
> ./see "int x [ ]"
```

```

GhasType (
  GString "x" ) (
  Garr
    GnoSize (
      Gintegral
        Gsigned
        Gint ) )

```

- 1 Introduction
- 2 Parsing C Types: An Extended Example
- 3 Using GF With Other Languages
- 4 Expressing Semantic Properties**
- 5 Using Libraries
- 6 Coda

# Declarations With Valid Size

```
abstract CType = {  
  
  cat Decl;  
  fun hasType : String -> CType HasSize -> Decl;
```

Restrict which types may appear in declarations.

# Declarations With Valid Size

```
abstract CType = {  
  
  cat Decl;  
  fun hasType : String -> CType HasSize -> Decl;  
  
  cat Size;  
  fun HasSize, NoSize : Size;
```

Dependent type: the type mentions a value.

# Declarations With Valid Size

```
abstract CType = {  
  
  cat Decl;  
  fun hasType : String -> CType HasSize -> Decl;  
  
  cat Size;  
  fun HasSize, NoSize : Size;  
  
  cat CType Size;
```

Declaration of CType similar to GADT.

# Sizes for Types

**fun**

```
float, double : CType HasSize;  
void : CType NoSize;
```

Each constructor specifies if it has a valid size.

# Sizes for Types

**fun**

```
float, double : CType HasSize;
```

```
void : CType NoSize;
```

```
ptr : (s : Size) -> CType s -> CType HasSize;
```

```
funT : (s : Size) -> CType s -> CType NoSize;
```

Pointers always have a size, independent of what they point to.

# Sizes for Types

## fun

```
float, double : CType HasSize;
```

```
void : CType NoSize;
```

```
ptr : (s : Size) -> CType s -> CType HasSize;
```

```
funT : (s : Size) -> CType s -> CType NoSize;
```

Dependent function: type of result depends on value of input.

# Sizes for Array Types

```
cat Dim Size;
```

```
fun
```

```
  arr      : (s : Size) -> Dim s -> CType HasSize -> CType s;
```

```
  noSize  : Dim NoSize;
```

```
  ofSize  : Int -> Dim HasSize;
```

# Changes to Concrete Syntax

**lin**

```
arr _      = postfix;  
funT _     = postfix ("(" ++ ")");  
ptr _ t    = case t.prec of { ...  
  
...}
```

- Almost no change, just ignore size parameters.
- No need to store them because they are implicit in notation.

# Example

```
import CTypeC.gf

ps -lexcode "char p[10]" | parse
> hasType "p" (arr HasSize (ofSize 10)
              (integral signed char))

ps -lexcode "void p[10]" | parse
> no trees found
```

# Different Styles of Dependent Types

- Parameterizing `type constructors` by properties works well.

```
cat CType Size;
```

```
fun void : CType NoSize;
```

# Different Styles of Dependent Types

- Parameterizing `type constructors` by properties works well.

```
cat CType Size FunRes;
```

```
fun void : CType NoSize CanReturn;
```

- But, sometimes it does not scale for many properties.
  - and array cannot be returned from functions,

# Different Styles of Dependent Types

- Parameterizing **type constructors** by properties works well.

```
cat CType Size FunRes FunArg;
```

```
fun void : CType NoSize CanReturn NoArg;
```

- But, sometimes it does not scale for many properties.
  - and array cannot be returned from functions,
  - and unsized arrays are OK in parameter declarations.

# Different Styles of Dependent Types

- Parameterizing **type constructors** by properties works well.

```
cat CType Size FunRes FunArg;  
fun void : CType NoSize CanReturn NoArg;  
  
hasType : String -> (t : CType)  
         -> ValidDeclType t -> Decl;
```

- Alternative: parameterize **value** constructors by proofs of properties.

# Different Styles of Dependent Types

- Parameterizing **type constructors** by properties works well.

```
cat CType Size FunRes FunArg;  
fun void : CType NoSize CanReturn NoArg;  
  
hasType : String -> (t : CType)  
         -> ValidDeclType t -> Decl;
```

- Alternative: parameterize **value** constructors by proofs of properties.
- Implicit proofs in documents are harder to recover.
- GF is still improving in this area.

- 1 Introduction
- 2 Parsing C Types: An Extended Example
- 3 Using GF With Other Languages
- 4 Expressing Semantic Properties
- 5 Using Libraries**
- 6 Coda

# English Notation Using Libraries

```
concrete CTypeEng1 of CType =  
open SyntaxEng, ParadigmsEng in {
```

Reuse predefined opers and concrete syntax.

# English Notation Using Libraries

```

concrete CTypeEng1 of CType =
open SyntaxEng, ParadigmsEng in {

```

## lincat

```

Decl      = Cl;  -- Clause (sentence)
CType     = CN;  -- Common noun
Sign      = A;   -- Adjective
Integral  = CN;  -- Common noun
Dim       = Num; -- Number determining element

```

Predefined grammatical categories.

# English Notation Using Libraries

## lin

```
short      = mkCN (mkA "short") (mkN "integer");  
int        = mkCN (mkN "integer");  
long       = mkCN (mkA "long") (mkN "integer");
```

Predefined syntax for various grammatical constructs.

# English Notation Using Libraries

## lin

```
short    = mkCN (mkA "short") (mkN "integer");  
int      = mkCN (mkN "integer");  
long     = mkCN (mkA "long") (mkN "integer");
```

Functions may be overloaded.

# English Notation Using Libraries

**lin**

```
short      = mkCN (mkA "short") (mkN "integer");
```

```
int       = mkCN (mkN "integer");
```

```
long      = mkCN (mkA "long") (mkN "integer");
```

```
arr n t   = mkCN (mkN "array")
            (mkRS (mkRC1 which_RP
                  (mkVP (mkV2 "contain")
                        (mkNP a_Quant n t)))));
```

...

```
}
```

Predefined standard words.

# English Notation Using Libraries

## lin

```

short      = mkCN (mkA "short") (mkN "integer");
int        = mkCN (mkN "integer");
long       = mkCN (mkA "long") (mkN "integer");

arr n t    = mkCN (mkN "array")
              (mkRS (mkRC1 which_RP
                    (mkVP (mkV2 "contain")
                          (mkNP a_Quant n t)))));
...
}

```

Functions adjust words to follow grammar.

# Example

```
import ../CTypeC.gf CTypeEng1.gf
```

```
ps -lexcode "int x[1024]"
```

```
| parse -lang=CTypeC
```

```
| linearize -unlextext -lang=CTypeEng1
```

> It is an array which contains signed integers

- 1 Introduction
- 2 Parsing C Types: An Extended Example
- 3 Using GF With Other Languages
- 4 Expressing Semantic Properties
- 5 Using Libraries
- 6 Coda**

# Summary

- The abstract syntax captures basic semantic ideas.
- There may be multiple concrete notations to express the ideas.
- By using dependent types we can avoid some meaningless expressions.
- GF may be used from other languages (e.g., Haskell, JavaScript).
- GF has a rich library for working with natural language.

# Useful Links

- Main GF site (tutorial, documentation, publications)  
<http://www.grammaticalframework.org/>
- Developers (bug tracker, links to repo)  
<http://code.google.com/p/grammatical-framework/>
- Discussion and questions:  
<http://groups.google.com/group/gf-dev>