Morphological Analysis
Functional Morphology

Daniel Zeman

http://ufal.mff.cuni.cz/course/npfl094
Functional Programming

• Functional programming languages
  – Stress the mathematical perception of functions
    • Strictly mapping some input on some output
    • No side effects, no dependence on the current state of the whole program
    • Program does not have state. Nothing like first \( a := 5 \), later on \( b := a+3; a := c \). Instead, we declare how the output relates to the input. If you say that \( a = 5 \), this statement is valid throughout the program.
      – If the same function is called on the same input twice, it is guaranteed that the output will be same as well.
  – Early functional programming language: LISP (e.g. macros in the GNU Emacs editor)
    – Caml
    – Haskell
Functional Morphology

– Gérard Huet, INRIA, France
  • Caml functional language
  • Zen CL Toolkit
  • Sanskrit morphology
– Chalmers Tekniska Högskola, Göteborg, Sweden
  • Haskell
    – Functional Morphology (FM) library by Markus Forsberg
  • Grammatical Framework (GF)
    – Functional programming language aimed at NLP
Forsberg’s FM for Haskell

- Haskell is a general-purpose functional language
  - http://tryhaskell.org/
- Functional morphology (FM, by Markus Forsberg & Aarne Ranta) is a library for Haskell
  - http://www.cse.chalmers.se/alumni/markus/FM/

- Unfortunately not maintained any more.
- Fortunately GF provides similar functionality.
Characteristics of FM

• Motivation: let linguists themselves code the morphology
  – Make description as simple and natural as possible
  – Minimize the necessity for programmer’s training
    • To start a new language, one needs to know *something* about Haskell
    • To add new words to existing language, no programming skills needed!
  – Functions and algebraic types: higher level of description than untyped regular expressions

• Library part implemented as a combination of multiple *tries* (recall Kimmo lexicons)
• Can be exported to the format of XFST (mainstream finite-state approach)
Characteristics of FM

- Core concept: paradigms (inflection tables)
- Inflection is defined as a function
- All approaches so far were centered around morphemes
  - Prefixes, stems and suffixes were all in lexicon and bore some meaning (lexical or grammatical)
  - A word was composed of morphemes
  - A word’s meaning was a composition of the morphemes’ meanings
- Now: stem + function
  - Only stems are lexicon units
  - Example of function: how to change a stem to get a plural form?
Paradigm Function

“A paradigm function is a function which, when applied to the root of a lexeme L paired with a set of morphosyntactic properties appropriate to L, determines the word form occupying the corresponding cell in L’s paradigm.”

Example:
Latin Paradigm *rosa* (*rose*)

<table>
<thead>
<tr>
<th>Case</th>
<th>Singular</th>
<th>Plural</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominative</td>
<td><em>rosa</em></td>
<td><em>rosae</em></td>
</tr>
<tr>
<td>Vocative</td>
<td><em>rosa</em></td>
<td><em>rosae</em></td>
</tr>
<tr>
<td>Accusative</td>
<td><em>rosam</em></td>
<td><em>rosas</em></td>
</tr>
<tr>
<td>Genitive</td>
<td><em>rosae</em></td>
<td><em>rosarum</em></td>
</tr>
<tr>
<td>Dative</td>
<td><em>rosae</em></td>
<td><em>rosis</em></td>
</tr>
<tr>
<td>Ablative</td>
<td><em>rosa</em></td>
<td><em>rosis</em></td>
</tr>
</tbody>
</table>
Paradigm Table in GF

table {
  Sing Nom => "rosa" ;
  Sing Voc => "rosa" ;
  Sing Acc => "rosam" ;
  Sing Gen => "rosae" ;
  Sing Dat => "rosae" ;
  Sing Abl => "rosa" ;
  Plur Nom => "rosae" ;
  Plur Voc => "rosae" ;
  Plur Acc => "rosas" ; ...
  ... Plur Abl => "rosis" }

Paradigm Function in GF

\begin{verbatim}
oper nounRosa : Str -> 
   {s: Number => Case => Str} = \rosa -> { 
   s = table { 
      Sing Nom => rosa ;
      Sing Voc => rosa ;
      Sing Acc => rosa + "m" ;
      Sing Gen => rosa + "e" ; …
      ... Plur Acc => rosa + "s" ;
      Plur Gen => rosa + "rum" ... 
      ... Plur Abl => (init rosa) + "is" };
\end{verbatim}
Abstract Grammar

abstract Grammar = {
  flags startcat = NOUN ;
  cat
    NOUN ;

  fun
    Woman,
    Mother,
    Meadow : NOUN ;

};
Concrete Grammar and Linearization

concrete GrammarZh of Grammar = {

lincat
    NOUN = {s : Str} ;

lin
    Woman = {s = "女人"} ;
    Mother = {s = "母亲"} ;
    Meadow = {s = "草地"} ;
    ...
}

Concrete Grammar and Linearization

concrete GrammarEn of Grammar = {

lincat
   NOUN = {s : Number => Str} ;

lin
   Woman = mkNoun "woman" "women" ;
   Mother = regNoun "mother" ;
   Meadow = regNoun "meadow" ;
   ...
}

Concrete Grammar and Linearization

concrete GrammarCs of Grammar = {

lincat
  NOUN = {s : Number => Case => Str} ;

lin
  Woman = nounZena "žena" ;
  Mother = nounZena "matka" ;
  Meadow = nounZena "louka" ;
...  
}

Paradigm Function

```plaintext
oper nounZena : Str -> 
{s: Number => Case => Str}
= \(\text{\textbackslash zena} \rightarrow\)
let zen : Str = (init zena) in 
{s = table { 
Sing Nom => zena ;
Sing Gen => zen + "y" ;
Sing Dat => zen + "e" ;
Sing Acc => zen + "u" ;
Sing Voc => zen + "o" ; ...
... Plur Ins => zen + "ami" }};
```
Paradigm Function

```haskell
oper nounZena : Str ->
{ s: Number => Case => Str }
= \zena ->
let zen : Str = (init zena) in
{ s = table {
  Sing Nom => zena ;
  Sing Gen => zen + "y" ;
  Sing Dat => mkDatLoc zen ;
  Sing Acc => zen + "u" ;
  Sing Voc => zen + "o" ; ...
  ... Plur Ins => zen + "ami" }
};
```
Phonological Changes

```haskell
oper mkDatLoc : Str -> Str =
\zen ->
case zen of {
  x + "k"  => x + "ce" ; -- matce
  x + "h"  => x + "ze" ; -- dráze
  x + "ch" => x + "še" ; -- sprše
  x + "r"  => x + "ře" ; -- káře
  _ + ("d"|"t"|"n"|"b"|"f"|"m"|"p"|"v")
     => zen + "ě"  ; -- ženě, vládě, mátě, bábě, …
  _    => zen + "e"  -- hale, base, koze
}
```

Pattern matching

Default “catch-all”
### Slightly Deviating Paradigm Artemis (Greek → Czech)

<table>
<thead>
<tr>
<th>Case</th>
<th>Greek</th>
<th>Czech</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominative</td>
<td>Artemis</td>
<td>žena “woman”</td>
</tr>
<tr>
<td>Genitive</td>
<td>Artemidy</td>
<td>ženy</td>
</tr>
<tr>
<td>Dative</td>
<td>Artemidě</td>
<td>ženě</td>
</tr>
<tr>
<td>Accusative</td>
<td>Artemidu (−s)</td>
<td>ženu</td>
</tr>
<tr>
<td>Locative</td>
<td>Artemidě</td>
<td>ženě</td>
</tr>
<tr>
<td>Instrumental</td>
<td>Artemidou</td>
<td>ženou</td>
</tr>
</tbody>
</table>
Inheritance: Slightly Deviating Paradigm *Artemis* (Greek → Czech)

```csharp
oper nounArtemis : Str ->
  {s: Number => Case => Str}
  \artemis -> let artemida :
  {s: Number => Case => Str}
  = nounZena ((init artemis) + "da") in
  { s = table {
    Sing => table {
      (Nom|Voc) => artemis ;
      c => artemida.s ! Sing ! c } ;
    Plur => table {
      c => artemida.s ! Plur ! c }
  } } ;
```
Parametric Types

```
param
    Gender = Masc | Fem ;
    Number = Sing | Plur ;

oper
    Noun : Type = {
        s: Number => Str;
        g: Gender
    } ;
    Adj : Type = {
        s: Gender => Number => Str
    } ;
```
Free Variation

```plaintext

```table
{ ... Plur Nom => pan + ("i"|"ové") ;
  ...
} ;
```

```table
{ ... Imper => variants {} ; ... } ;
```

• Drawback: variation in one branch means the entire table is duplicated / erased. The following no longer work as expected:
  - `linearize -table Word`
  - `morpho_analyze "word"`

• Alternative: new parameter (feature) `Variant`
Arabic Morphology

```plaintext
oper
word : (pattern, root : Str) -> Str ;

yaktubu = word "yaFCuLu" "ktb" ;

Root : Type = {F,C,L : Str} ;
Pattern : Type = Root -> Str ;
Filling : Type = {F,FC,CL,L : Str} ;

```
Arabic Morphology

```haskell
getRoot : Str -> Root = \s -> case s of {
  F@? + C@? + L@? => {F=F; C=C; L=L} ;
}

getPattern : Str -> Pattern = \s -> case s of {
  F + "F" + FC + "C" + CL + "L" + L =>
    fill {F=F; FC=FC; CL=CL; L=L} ;
  _ => Predef.error("cannot get pattern from" ++ s) } ;

word : (patt, root : Str) -> Str = \p,r -> getPattern p (getRoot r) ;
```
Exercise: German Umlauts

• First try GrammarCs (sample Czech paradigms)
  – See the lab web for instructions
  – Also inspect the source code

• Then open GrammarDe
  – Write oper umlaut : Str -> Str that can be called on the stem if stem vowel has to be umlauted
Batch Processing

• Script of GF commands:
  
  ```
  import GrammarEn.gf
  ma "the cats were chasing the dogs"
  ma "there is one more sentence"
  ```

• Run GF with the script:

  ```
  gf --run < myscript.gfs > analyzed.gfma
  ```
CAML/FM/GF Applications

- Sanskrit (Gérard Huet)
- Swedish, Spanish, Russian, Italian, Latin
  - Originally by Markus Forsberg in FM (defunct)
  - GF: many more languages
- Muhammad Humayoun (محمد بمايون):
  - Urdu morphology in FM/GF (2006)
  - Punjabi morphology in GF (2010)
Elixir FM

• Otakar Smrž:
  – Functional Morphology of Arabic (Elixir FM; PhD thesis 2006–2010; in Haskell)
    • [http://quest.ms.mff.cuni.cz/cgi-bin/elixir/index.fcgi](http://quest.ms.mff.cuni.cz/cgi-bin/elixir/index.fcgi)
GF Applications: More than Morphology!

- Interlingua-based translation system
  - Cf. TectoMT, Apertium
- Simple domains:
  - Tourist phrasebooks
  - Language learning
- GF API available for Haskell, Java, JavaScript
GF is More than Morphology!

- Grammar of constituents (mildly context-sensitive)
  - Model entire clauses
  - Multilingual: abstract syntax vs. many concrete syntaxes
- See the demo