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
Grammatical Framework (GF)

• Functional language tailored for NLP tasks
• Applications:
  – Muhammad Humayoun (محمد بمايون):
    • Urdu morphology in GF (2006)
    • Punjabi morphology in GF (2010)
      – http://www.lama.univ-savoie.fr/~humayoun/punjabi/
Forsberg’s FM for Haskell

- [http://www.cse.chalmers.se/alumni/markus/FM/](http://www.cse.chalmers.se/alumni/markus/FM/)
- Haskell is a general-purpose functional language
  - [http://tryhaskell.org/](http://tryhaskell.org/)
- **Functional morphology** (FM, by Markus Forsberg & Aarne Ranta) is a library for Haskell
  - Can be viewed as a domain-specific embedded language with Haskell as host language
- As if a morphology-related part of GF had been re-implemented in Haskell
- Some operations (e.g. on lists) are more easily described in Haskell than in GF
Forsberg’s FM for Haskell

- [http://www.cse.chalmers.se/alumni/markus/FM/](http://www.cse.chalmers.se/alumni/markus/FM/)
- Download together with Latin morphology
- Prepared for Linux (configure, make, sudo make install)
- Should work in Haskell Platform for Windows too
  - At least in theory (Haskell sources are distributed)
  - I have not been able to make it work there yet
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
  - Lexicon, rules and the actual implementation all together in one framework (the Haskell language!)
- Based on Huet’s Zen toolkit for Sanskrit
- 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>
Function Syntax in Haskell

\[
\text{addS} :: \text{String} \rightarrow \text{String} \\
\text{addS} \text{ cat} = \text{cats} \\
\quad \text{where} \\
\quad \text{cats} = \text{cat} ++ \text{"s"}
\]

Function with two parameters
= function that returns function

\[
\text{drop} :: \text{Int} \rightarrow [\text{a}] \rightarrow [\text{a}]
\]
The Paradigm in Haskell

\[
data \ \text{Case} = \text{Nom} \mid \text{Voc} \mid \text{Acc} \mid \text{Gen} \mid \ldots
\]
\[
data \ \text{Number} = \text{Sg} \mid \text{Pl}
\]
\[
data \ \text{NounForm} = \text{NounForm} \ \text{Case} \ \text{Number}
\]

Like a tag. Example value is “\text{NounForm Nom Sg}”.

\[
type \ \text{Noun} = \text{NounForm} \to \text{String}
\]

Every noun is a function that maps tags to forms.

All are defined for the same domain.

Each has its own value range.

\[
\text{rosaParadigm} :: \text{String} \to \text{Noun}
\]

Example paradigm function.

Input: a lemma. Output: its noun function, i.e. table of forms.
The Paradigm in Haskell

data Case = Nom | Voc | Acc | Gen | Dat | Abl

data Number = Sg | Pl

data NounForm =
    NounForm Case Number

type Noun = NounForm -> String

rosaParadigm :: String -> Noun
    (NounForm c n) =
    let
        rosae = rosa ++ "e"
        rosis = init rosa
        ++ "is"
    in
        case n of
            Sg -> case c of
                Acc -> rosae
                Gen -> rosae ++ "m"
                _   -> rosae
            Pl -> case c of
                Nom -> rosae
                Voc -> rosae
                Acc -> rosae ++ "s"
                Gen -> rosae
                _   -> rosis
Words Belonging to Paradigm

rosa, causa, barba :: Noun

rosa  = rosaParadigm "rosa"
causa = rosaParadigm "causa"
barba = rosaParadigm "barba"

Examples:
causa (NounForm Gen Pl)

causarum

rosaParadigm "xxxxxxxa" (NounForm Abl Pl)

xxxxxxxis
Using the Paradigm for Slightly Deviating Noun *dea* (*goddess*)

```haskell
dea :: Noun
dea nf =
    case nf of
        NounForm Dat Pl -> dea
        NounForm Abl Pl -> dea
        _ -> rosaParadigm dea nf
    where dea = "dea"
```

10.12.2010

http://ufal.mff.cuni.cz/course/npfl094
Turning a Function into a Table

• A paradigm function is good for *generating* forms
• FM function `table` compiles a function into lookup tables and further to tries
  – Then we can also *analyze* word forms

```haskell
table :: Param a => (a -> Str) -> [(a,Str)]
table f = [(v, f v) | v <- values]
```
Extended String $\texttt{Str}$

- Abstract type implemented as list of strings: $\texttt{[String]}$
- Free variation, e.g. singular dative of $\textit{domus}$ (home) can be $\textit{domui}$ or $\textit{domo}$
  - list of two strings
- Missing forms, e.g. $\textit{vis}$ (force) is defective in that it doesn’t have singular vocative, genitive and dative
  - an empty list
Example: Umlaut in German Plural

```haskell
findStemVowel :: String -> (String, String, String)
findStemVowel sprick
  = (reverse rps, reverse i, reverse kc)
  where
    (kc, irps) = break isVowel $ reverse sprick
    (i, rps)   = span isVowel $ irps

umlaut :: String -> String
umlaut man = m ++ mkUm a ++ n
  where
    (m, a, n) = findStemVowel man
    mkUm v = case v of
      "a"  -> "ä"
      "o"  -> "ö"
      "u"  -> "ü"
      "au" -> "äu"
      _    -> v

baumPl :: String -> String
baumPl baum = umlaut baum ++ "e"
```

Batch Processing

- Haskell programs can be interpreted or compiled
- Haskell platform / the GHC compiler
- Interpreting in interactive mode
  - Run ghci, load Haskell source, then call functions
- Standalone: define function main().
- Run script in Haskell without compiling it:
  - runghc program.hs
Existing Applications

• Markus Forsberg:
  – Swedish, Spanish, Russian, Italian, Latin
    • http://www.cse.chalmers.se/alumni/markus/phd2007_print_version.pdf

• 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