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 ہمایوں):
    - Urdu morphology in GF (2006)
      - http://www.lama.univ-savoie.fr/~humayoun/UrduMorph/
    - 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/
- Haskell is a general-purpose functional language
  - 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/
- 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."*

(Gregory T. Stump: *Inflectional Morphology. A Theory of Paradigm Structure*. Cambridge Studies in Linguistics, Cambridge University Press, Cambridge, UK, 2001, p. 32)

Example:
Latin Paradigm *rosa (rose)*

	Singular	**Plural**
Nominative	*rosa*	*rosae*
Vocative	*rosa*	*rosae*
Accusative	*rosam*	*rosas*
Genitive	*rosae*	*rosarum*
Dative	*rosae*	*rosis*
Ablative	*rosa*	*rosis*

Function Syntax in Haskell

declaration

```
addS :: String -> String
addS cat = cats
  where
    cats = cat ++ "s"
```

description

Function with two parameters
= function that returns function
e.g.

```
drop :: Int -> [a] -> [a]
```

The Paradigm in Haskell

```
data Case = Nom | Voc | Acc | Gen | …
data Number = Sg | Pl
data NounForm = NounForm Case Number
```

Like a tag. Example value is "NounForm Nom Sg".

```
type Noun = NounForm -> String
```

Every noun is a function that maps tags to forms.
All are defined for the same domain.
Each has its own value range.

```
rosaParadigm :: String -> 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
rosaParadigm rosa
   (NounForm c n) =
   let
     rosae = rosa ++ "e"
     rosis = init rosa
             ++ "is"
   in
     case n of
       Sg -> case c of
         Acc -> rosa
                ++ "m"
         Gen -> rosae
         Dat -> rosae
         _   -> rosa
       Pl -> case c of
         Nom -> rosae
         Voc -> rosae
         Acc -> rosa
                ++ "s"
         Gen -> rosa
                ++ "rum"
         _   -> 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 “xxxxxxa” (NounForm Abl Pl)
  xxxxxxis
```

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

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

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

```
table :: Param a => (a -> Str) ->
  [(a,Str)]
table f = [(v, f v) | v <- values]
```

Extended String `Str`

- Abstract type implemented as list of strings: **`[String]`**
- Free variation, e.g. singular dative of *domus (home)* can be *domui* or *domo*
  - list of two strings
- Missing forms, e.g. *vis (force)* is defective in that it doesn't have singular vocative, genitive and dative
  - an empty list

Example: Umlaut in German Plural

```
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