Functional Morphology

doing linguistics in Haskell

Markus Forsberg, Aarne Ranta

ICFP 2004

Snowbird, Utah

What is a morphology?

- Think of a **dictionary for a natural language** stored in a computer
- How is a dictionary organized?
  Normally we don't find all word forms, but:
  - A set of **inflection tables**, also called **paradigms**
  - A list of **entries** with a **pointer** to a **inflection table**.
    The pointer can be *enough grammar information* so that you can "point for yourself".

Why do we need a morphology?
Some examples

- **Machine translation**, e.g.
  - To retrieve the grammatical information about the words
  - To retrieve all possible analyses for the disambiguation phase
  - To generate a particular word form in a target language
- **Information retrieval**
  Searching for: *cars*
  you may also be interested in: *car*, *cars*, *car's cars'*
  but not: *cart*, *carisma*, *Carter*
- **Language education**
  - language quizzes
  - study material resources

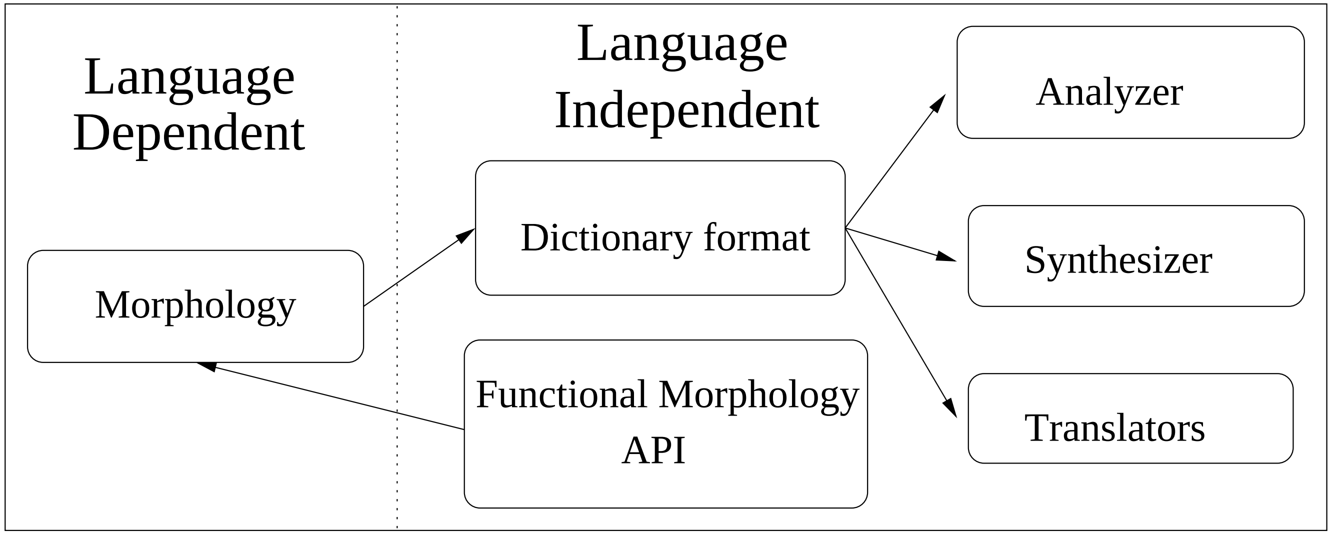
Earlier implementations of morphology

- More or less hand-written databases and full-form lexica have been around since the 1950's
- Proprietary systems in proprietary formats: XFST (Xerox) current state of the art, finite state technology
- Huet's Zen toolkit and Sanskrit morphology in CAML is a substantial example of the functional methodology and provides algorithms that we have used in our work

Three views of Functional Morphology

- A **methodology** for developing a morphology in a typed functional language.
- An **embedded domain-specific language** in Haskell for morphology development.
- A collection of **morphology implementations**.

Overview of the system



Translator to:

- XFST and LexC
- GF (Grammatical Framework)
- XML
- SQL
- Full-form lexicon, tables and LaTeX.

Morphological analysis of text

```
...
[ <gud>
1. gud (3309) Substantiv - Sg Indef Nom - Utr
]

[ <såg>
1. såg (4693) Substantiv - Sg Indef Nom - Utr
2. se (198) Verb - Pret Conj Act -
3. se (198) Verb - Pret Ind Act -
]

[ <att>
1. att (148) Infinitivmärke - Invariant -
2. att (94) Subjunktion - Invariant -
]
...
```

Morphological synthesis

```
********************************************
* Swedish Morphology                       *
********************************************
* Functional Morphology v1.00              *
* (c) Markus Forsberg & Aarne Ranta 2004   *
* under GNU General Public License.        *
********************************************

[Synthesiser mode]

Enter a Swedish word in any form.

Type 'c' to list commands.

Dictionary loaded: DF = 17209 and WF = 228262.

> programmet

program - Substantiv - Neutr

Sg Indef Nom: program
Sg Indef Gen: programs
Sg Def Nom: programmet
Sg Def Gen: programmets
Pl Indef Nom: program
Pl Indef Gen: programs
...
```

A morphology in FM

- The backbone of FM consists of three type classes: `Param`, `Dict` and `Language`. Enable code resuse and generic algoritms for analysis, synthesis and code generation.
- Fundamentally, a morphology in FM has:
  - A **type system**: defines all word classes and the parameters belonging to them.
  - An **inflection machinery**: defines all possible inflection tables (paradigms) for all word classes.
  - A **lexicon**: lists all words in the target language with their paradigms.

Nouns in Latin

- The Latin noun *rosa*, a feminine noun in the first declension.

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

- Think of *rosa* as an example word for the first declension paradigm.

Nouns in Latin: type system

(Inflectional) parameter types as algebraic data types:

```
data Case   =
 Nominative | Genitive | Dative |
 Accusative | Ablative | Vocative
   deriving (Show,Eq,Enum,Ord,Bounded)

data Number = Singular | Plural
   deriving (Show,Eq,Enum,Ord,Bounded)

data NounForm = NounForm Number Case
   deriving (Show,Eq)
```

Nouns in Latin also have a *inherent* parameter: `Gender`. Nouns in Latin **have** a gender, they are not inflected in gender.

```
  data Gender = Feminine  | Masculine | Neuter
   deriving (Show,Eq,Enum,Ord,Bounded)
```

Type hierarchy

- A more complicated word class, latin verbs

```
data VerbForm =
    Indicative Person Number Tense Voice   |
    Infinitive TenseI Voice                |
    ParticiplesFuture  Voice               |
    ParticiplesPresent                     |
    ParticiplesPerfect                     |
    Subjunctive Person Number TenseS Voice |
    ImperativePresent Number Voice         |
    ImperativeFutureActive  Number PersonI |
    ImperativeFuturePassiveSing PersonI    |
    ImperativeFuturePassivePl              |
    GerundGenitive                         |
    GerundDative                           |
    GerundAcc                              |
    GerundAbl                              |
    SupineAcc                              |
    SupineAblative
```

- 147 cases, compared with the product of 1260 cases

rosa as a Haskell table

A first attempt of describing a paradigm.

```
rosaParadigm :: String → [(NounForm,String)]
rosaParadigm rosa =
    [
      (NounForm Singular Nominative, rosa),
      (NounForm Singular Vocative, rosa),
      (NounForm Singular Accusative, rosa ++ "m"),
      (NounForm Singular Genitive, rosa ++ "e"),
      (NounForm Singular Dative, rosa ++ "e"),
      (NounForm Singular Ablative, rosa),
      (NounForm Plural Nominative, rosa ++ "e"),
      (NounForm Plural Vocative, rosa ++ "e"),
      (NounForm Plural Accusative, rosa ++ "s"),
      (NounForm Plural Genitive, rosa ++ "arum"),
      (NounForm Plural Dative, rosis),
      (NounForm Plural Ablative, rosis)
    ]
  where rosis = tk 1 rosa ++ "is"
```

Not very nice ... Difficult to express linguistic abstraction and error-prone.

Finite parameters, towards finite functions

The class `Param` provide a constant `values` that enumerates all values in a parameter type.

```
class (Eq a, Show a) ⇒ Param a where
  values  :: [a]
  -- and some default definitions
```

```
 instance Param Case     ...
 instance Param Number   ...
 instance Param NounForm ...
```

Nouns as finite function

- A noun as a function — given a parameter it gives a word form.

```
type Noun = NounForm → String
```

- `NounForm` is an instance of `Param`, so a `Noun` is easily turned into a table:

```
table :: Param a ⇒ (a → String) → [(a,String)]
table f = [(v, f v) | v <- values]

nounTable :: Noun -> [(a,String)]
nounTable f = table f
```

rosa as a function

- The inflection table of *rosa* as a function.

```
rosaParadigm :: String → Noun
rosaParadigm rosa (NounForm n c) =
   case n of
    Singular → case c of
                 Accusative → rosa + "m"
                 Genitive   → rosae
                 Dative     → rosae
                 _          → rosa
    Plural   → case c of
                 Nominative  → rosae
                 Vocative   → rosae
                 Accusative → rosa ++ "s"
                 Genitive   → rosa ++ "rum"
                 _          → ros  ++ "is"
 where  rosae = rosa ++ "e"
        ros   = init rosa
```

Strings

- A word form in FM is actually a list of strings, not a single one.
- This to handle **non-existing forms** and **free variation** (many word forms may be possible for a particular set of inflectional parameters).

```
type Str = ...

mkStr :: String → Str

strings :: [String] -> Str

nonExist :: Str
```

vis, Non-existing forms

- *vis* is an example of a word that lacks some forms.

	Singular	**Plural**
Nominative	*vis*	*vires*
Vocative	-	*vires*
Accusative	*vim*	*vires*
Genitive	-	*virium*
Dative	-	*viribus*
Ablative	*vi*	*viribus*

Exceptions

- Exceptions are used to define paradigm in terms of other paradigms, or a lemma that is close to a particular paradigm.
- Exceptions in FM: **`excepts`**, **`missing`**, **`only`**, **`variants`**.

```
dea :: Noun
dea =
   (rosaParadigm "dea") 'excepts'
     [(NounForm Plural c, "dea") | c <- [Dative, Ablative]]
```

```
vis :: Noun
vis = (hostisParadigm "vis") 'missing'
        [NounForm Singular c | c <- [Vocative, Genitive, Dative]]
```

Translation to the language-independent dictionary

- Every morphology is translated into the language-independent dictionary.
- This can be done almost automatically — we already know how to create an inflection table, but we need some additional information.

```
class Param a ⇒ Dict a where
  dictword      :: (a → Str) → String       -- lemma
  category      :: (a → Str) → String       -- word class
  defaultAttr   :: (a → Str) → Attr         -- composite analysis
  attrException :: (a → Str) → [(a,Attr)]   -- composite analysis

 instance Dict NounForm where
   category _ = "Noun"
```

- Note: the use of a function avoids the inconvenience of having to supply an object in `a`.

Dictionary: interface functions

- A dictionary consists of a set of `Entry`:s.
- A couple of interface functions about nouns in general:

```
entryI :: Dict a => (a -> Str) -> [Inherent] -> Entry

noun :: Noun → Gender → Entry
noun n g = entryI n [prValue g]

masculine :: Noun → Entry
masculine n = noun n Masculine

feminine :: Noun → Entry
feminine n = noun n Feminine

neuter :: Noun → Entry
neuter n = noun n Neuter
```

Dictionary: interface functions

- We continue by defining a few interface functions for a couple of paradigms.

```
d1rosa :: String → Entry
d1rosa = feminine . decl1rosa

d2servus :: String → Entry
d2servus = masculine . decl2servus

d2donum :: String → Entry
d2donum s = neuter . decl2donum
```

The (internal) dictionary

```
latinDict :: Dictionary
latinDict =
 dictionary $
  [
   d1rosa    "rosa",
   d1rosa    "puella",
   d2servus  "servus",
   d2servus  "somnus",
   d2servus  "amicus",
   d2servus  "animus",
   d2servus  "campus",
   d2servus  "cantus",
   d2servus  "caseus",
   d2servus  "cervus",
   d2donum   "donum"
  ]
```

The internal and external dictionary

- The **internal** dictionary usually describes the closed word classes (conjunction, preposition etc) and the highly irregular cases. Compiles into the program.
- The **external** dictionary usually contains the open word classes (nouns, verbs, adjectives etc). Consists of an external file that lists paradigms and lemmas.

External dictionary

- Paradigm with the word in lemma form.
- (latin.lexicon)

```
d1rosa   rosa
d2servus servus
v1amare  amare
v1amare  portare
v1amare  demonstrare
v1amare  laborare
...
```

Analysis: trie

- The analysis in FM is done in the same manner as in Huet's Zen toolkit — we use a decorated trie as fundamental data structure.
- A trie is an acyclic, one-way-directed transducer, that can be built and run efficiently.
- We handle composite analysis by permitting cycles over the trie.

Composite analysis in FM

- Forms are given attribute values in the `Attr` type, that describes how they can be composed.
- The developer provides a boolean function that describes which compositions are valid, e.g. :

```
composeLatin :: [Attr] → Bool
```

- The default is that no word compositions is valid — words can only appear by themselves.

Composite forms: example

- Consider the question particle *ne* in Latin, which can be added as a suffix to any word in Latin, and thereby put the word in question.
- We describe with our attribute values and attribute function that *ne* only can occur as a suffix.
- If we now analyze the word *servumne*, we would get the following:

```
[ <servumne>
  Composite:
  servus Noun - Singular Accusative - Masculine
  |  # ne Particle - Invariant -]
```

The runtime system

The `Language` class consists of functions for the runtime system. All functions have a default definition.

```
class Show a ⇒ Language a where
  name        :: a → String
  dbaseName   :: a → String
  composition :: a → [Attr] → Bool
  env         :: a → String
  paradigms   :: a → Commands
  internDict  :: a → Dictionary

data Latin = Latin
   deriving Show

instance Language Latin where
  ...
```

Lexicon Extraction (the `extraction` tool)

- The Word and Paradigm representation in FM open up the door for automatic lexicon extraction.
- The idea is simple: let a set of affixes identify a particular paradigm, and use a Trie data structure to search for new words in a corpus.
- A great help in lexicon building, but ... the problem often has no solution, and manual checking is necessary.

Example of a paradigm file: Swedish

```
s1:  ap/a apor 3
s2:  al alen alar 2
s2:  cyk/el cykeln cyklar 4
s3:  oas oaserna 3
s4:  id/e idet idena 3
s5:  ris riset 2
s5:  öv/are övarna 5
aReg:  dyr dyrt dyra (dyraste|dyrare|dyrast) 2
aReg:  gir/ig girigt 4
aReg:  fag/er fagra 4
aReg:  gal/en galet galna 4
v1:  an/a (anar|anade|anat) 3
v2:  ös/a öser (öste|öst) 3
v3:  sy syr sydde 2
```

Lexicon extracted from the Swedish Bible

Found 1995 lemmas, e.g.

```
s1 möda
v1 möda
v1 mörda
aReg mörk
v2 mörka
s5 möt
v2 möta
v1 nagla
aReg naken
s5 namn
s3 nasir
aReg nedbruten
```

Results

- Swedish - 17 000 lemmas
- Spanish - 12 000 lemmas
  (Master thesis of Inger Andersson, Therese Söderberg)
- Russian - 9 000 lemmas
  (Master thesis of Ludmilla Bogavac)
- Italian - 5 000 lemmas
- Latin - tutorial language

Conclusion

- Functional Morphology has showned to be very *productive*.
- The use of Haskell gives access to powerful programming constructs that can be used to capture linguistic generalizations.
- Students with no previous Haskell experience have managed to produce substantial morphologies in FM.
- Functional Morphology is freely available under the GPL license, downloadable here:
  `http://www.cs.chalmers.se/~markus/FM`
  Some stuff is still in the cvs, but will soon appear on the webpage.