Functional Morphology
A Tutorial

Markus Forsberg
Department of Computing Science
Chalmers University of Technology
markus@cs.chalmers.se

October 20, 2004

1 Introduction

This document describes how to develop a morphology in Functional Morphology (abbreviated from now on as FM). The language we will consider, as a running example, is Latin, a highly inflected language that serves well to demonstrate the fundamentals of FM. The tutorial assumes that you have downloaded the source code `FM_LAT_1.0.tgz`.

To benefit from this tutorial, a basic knowledge in the functional language Haskell is required.

2 Overview

A morphology implementation in FM of Latin consists of the following modules in the functional language Haskell:

- Type system (`TypeLat.hs`).
- Paradigms as functions (`RulesLat.hs`).
- The dictionary with interface functions that denote entries in a dictionary (`BuildLat.hs`, `DictLat.hs`).
- An optional composite analysis (`AttrLat.hs`, `CompositeLat.hs`)
- The runtime system (`CommandsLat.hs`, `latin.lexicon`, `Main.hs`).

3 Step 1: Type system

3.0.1 Parameter types (`TypesLat.hs`

)

In grammars, words are divided into classes according to their orthographical similarity, such as having similar inflection patterns, and where they can occur and what role they play in a sentence. Examples of classes, the *part of speech*, are nouns, verbs, adjectives and pronouns.

Words in a class are attributed with a set of parameters that can be divided into two different kinds of categories: *inflectional* parameters and *inherent* parameters.

Parameters are best explain with an example. Consider the Latin noun *causa* (Eng. *cause*). It is inflected in number and case, i.e. number and case are the inflectional parameters. It also has a gender, which is an inherent parameter. The inflection of *causa* in plural nominative is *causae*, but it *has* feminine gender.

These parameters are described with the help of Haskell's data types. For example, to describe the parameters for Latin noun, the types `Gender`, `Case` and `Number` are introduced.

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

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

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

The inflectional parameter types `Case` and `Number` are combined into one type, `NounForm`, that describes all the inflection forms of a noun. Note that `Gender` is not part of the inflection types, it is an inherent parameter.

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

The parameter types of a language are language-dependent. A class `Param` for parameters has been defined, to make it possible to define language independent methods, i.e. implement generic algorithms.

```
  class (Eq a, Show a) ⇒ Param a where
   values   :: [a]
   value    :: Int → a
   value0   :: a
   prValue  :: a → String
   value n  = values !! n
   value0   = value 0
   prValue  = show
```

The most important method — the only one not defined by default — is `values`, giving the complete list of all objects in a `Param` type. The parameter types are, in a word, hereditarily finite data types: not only enumerated types but also types whose constructors have arguments of parameter types.

An instance of `Param` is easy to define for bounded enumerated types by the function `enum`.

```
enum :: (Enum a, Bounded a) ⇒ [a]
enum = [minBound .. maxBound]
```

The parameters of Latin nouns are made an instance of `Param` by the following definitions:

```
instance Param Gender  where values = enum
instance Param Case    where values = enum
instance Param Number  where values = enum
instance Param NounForm where
 values =
  [NounForm n c | n <- values ,
                  c <- values]
 prValue (NounForm n c) =
   unwords $ [prValue n, prValue c]
```

The default definition for `prValue` has been redefined for `NounForm` to remove the `NounForm` constructor. Usually, a more sophisticated printing scheme is preferred, using a particular *tag set*, i.e. adopting a standard for describing the parameters of words.

Latin nouns can now be defined as a finite function, from a NounForm to a `Str`. A `Str` is actually a list of string, to handle missing forms and variants.

```
type Noun = NounForm -> Str
```

The `Str` type is kept abstract, to enable a change of the representation. The abstraction function is called `strings`.

```
strings :: [String] → Str
string = id
```

The normal case is singleton lists, and to avoid the increased complexity of programming with lists of strings, we provide the `mkStr` function, that promotes a `String` to a `Str`.

```
mkStr :: String → Str
mkStr = (:[])
```

The description of missing cases is handled with the constant `nonExist`, which is defined as the empty list.

```
nonExist :: Str
nonExist = []
```

3.1 String operations

Functional Morphology provides a set of string operation functions that captures common phenomena in word inflections. Some of them are listed below to serve as examples.

The string operations cannot be quite complete, and a morphology implementer typically has to write some functions of her own, reflecting the peculiarities of the target language. These new functions can be supplied as an extended library, that will simplify the implementation of a similar language. The goal is to make the library so complete that linguists with little knowledge of Haskell can find it comfortable to write morphological rules without recourse to full Haskell.

Here is a sample of string operations provided by the library.

The Haskell standard functions `take` and `drop` take and drop prefixes of words. In morphology, it is much more common to consider *suffixes*. So the library provides the following dual versions of the standard functions:

```
tk :: Int → String → String
tk i s = take (max 0 (length s - i)) s

dp :: Int → String → String
dp i s = drop (max 0 (length s - i)) s
```

It is a common phenomenon that, if the last letter of a word and the first letter of an ending coincide, then one of them is dropped.

```
  (+?) :: String → String → String
  s +? e = case (s,e) of
            (_:_,c:cs) | last s == c → s ++ cs
            _ → s ++ e
```

More generally, a suffix of a word may be dependent of the last letter of its stem.

```
  ifEndThen :: (Char → Bool) → String → String
                             → String → String
  ifEndThen cond s a b = case s of
                          _:_ | cond (last s) → a
                          _ → b
```

A more language dependent function, but interesting because it is difficult to define on this level of generality with a regular expression, is the *umlaut* phenomenon in German, i.e. the transformation of a word's stem vowel when inflected in 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" → "u"
             "au" → "äu"
             _    → v
```

The plural form of *Baum*, can be describe with the function `baumPl`.

```
  baumPl :: String → String
  baumPl baum = umlaut baum ++ "e"
```

Applying the function `baumPl` with the string `"Baum"` computes to the correct plural form `"Bäume"`.

Obviously, the function *umlaut* is a special case of a more general *vowel alternation* function, that is present in many language, for instance, in English in the thematic alternation of verbs such as *drink-drank-drunk*:

```
vowAltern :: [(String,String)] → String → String
vowAltern alts man = m ++ a’ ++ n
  where
   (m,a,n) = findStemVowel man
   a’ = maybe a id $ lookup a alts
```

A general lesson from vowel alternations is that words are not just strings, but data structures such as tuples. If regular expressions are used, these data structures have to be encoded as strings with special characters used as delimiters, which can give rise to strange errors since there is no type checking.

4 Step 2: Paradigms as functions

4.1 `RulesLat.hs`

Let us start by considering the inflection table of `rosa` (Eng. rose), that illustrates the first declension of latin nouns. Words with the same inflection table are also said to be in the same *paradigm*.

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

The concept of inflection tables corresponds intuitively, in a programming language, to a list of pairs. Instead of using list of pairs, a functional counterpart of a table — a finite function could be used, i.e. a finite set of pairs defined as a function.

Defining the first declension illustrated by the inflection table of *rosa* is quite simply a translation from a table to a function, here in terms of another word in the first declension, *puella*.

```
decl1 :: DictForm -> Noun
decl1 puella (NounForm n c) =
    mkStr $
     case n of
      Singular -> case c of
                   Accusative -> puella ++ "m"
                   Genitive   -> puella ++ "e"
                   Dative     -> puellae
                   _          -> puella
      Plural   ->  case c of
                   Nominative -> puellae
                   Vocative   -> puellae
                   Accusative -> puella ++ "s"
                   Genitive   -> puella ++ "rum"
                   _          -> (tk 1 puella) ++ "is"
 where puellae = puella ++ "e"
```

The `RulesLat.hs` consists of functional definitions of a small amount of Latin paradigms.

4.2 Exceptions

Exceptions are used to describe paradigms that are similar to another paradigm, with the exception of one or more case. That is, instead of defining a completely new paradigm, we use the old definition only marking what is different. This is not only linguistically more satisfying, it saves a lot of work. Four different kinds of exceptions, `excepts`, `missing`, `only` and `variants`, are listed below.

The exception `excepts`, takes a finite function, or a paradigm in other words, and list of exceptions, and forms a new finite function with with exceptions included.

```
 excepts :: Param a ⇒ (a → Str) → [(a,Str)] → (a → Str)
```

The exception functions `missing` and `only` are used to express missing cases in a table; `missing` enumerates the cases with missing forms, and `only` is used for highly defective words, where it is easier to enumerate the cases that actually exists.

```
  missing :: Param a ⇒ (a → Str) → [a] → (a → Str)

  only :: Param a ⇒ (a → Str) → [a] → (a → Str)
```

An often occurring exception is additional variants, expressed with the function `variants`. That is, that a word is in a particular paradigm, but have more than one variant in one or more forms.

```
  variants :: Param a ⇒ (a → Str) → [(a,String)] → (a → Str)
```

5 Step 3: The dictionary

5.1 Interface functions: BuildLat.hs

Every language is defined with its own set of types, and to be able to define generic translations, we need to translate it into an intermediary format, called `Dictionary`. This process is more or less automatic by making part of speech types an instance of the Dict class.

Returning to the noun example, `NounForm` can be defined as an instance of the class `Dict` by giving a definition of the `category` function.

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

Given that `NounForm` is an instance of the `Dict` class, a function `noun` can be defined, that translates a `Noun` into an dictionary entry, including the inherent parameter `Gender`, and a function for every gender.

```
 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
```

Furthermore, we can define a set of *interface functions* that translates a dictionary word into a dictionary entry: `d2servus` (Eng. *servant, slave*), `d1puella` (Eng. *girl*) and `d2donum` (Eng. *gift, present*).

```
d2servus :: String -> Entry
d2servus = masculine . decl2servus

d1puella :: String -> Entry
d1puella = feminine . decl1puella

d2donum :: String -> Entry
d2donum s = neuter . decl2donum
```

Given these interface function, a dictionary with words can be created. Note that the function `dictionary` is an abstraction function that is presently defined as `id`.

5.2 Internal lexicon: `DictLat.hs`

The internal dictionary consists of a list of words with their paradigm.

```
latinDict :: Dictionary
latinDict =
 dictionary $
```

```
[
 d2servus  "servus",
 d2servus  "somnus",
 d2servus  "amicus",
 d2servus  "animus",
 d2servus  "campus",
 d2servus  "cantus",
 d2servus  "caseus",
 d2servus  "cervus",
 d2donum   "donum",
 feminine $ (d1puella "dea") `excepts`
   [(NounForm Plural c,"dea") | c <- [Dative, Ablative]]
]
```

6 Step 6: Composite analysis (optional)

FM offers the possibility to do composite analysis. By default, all words is just singleton words. To define composite analysis, we need to do two things: first define a set of attributes that describes what kind of composition property a particular part of speech or even a specific word form has. Secondly, a function that given a list of attribute values, decides if it is a valid decomposition.

6.1 Attributes: `AttrLat.hs`

The `Attr`, that describes a composition property, is simply a set of integer that must to be larger than 1. For the purpose of our example, we only define one attribute, `atS`, that describes that a specific form can only occur as a suffix to another word.

```
atS :: Attr
atS = 1
```

6.2 Legal composite forms: `CompositeLat.hs`

When we decompose a a string into a set of words, they are all tagged with an `Attr`. The job for the morphology developer is to say which sequences of `Attr` is actually legal decompositions.

In Latin, we have particles that can be added as suffixes to all words. We have, for example, the particle *ne*, that when added as a suffix to a word puts that word into question. We give that type of particle the attribute value `atS`, and defines that is a valid decomposition if the compound consists of two words, and the second one is a suffix and the first one is not.

```
latinDecompose :: [Attr] → Bool
latinDecompose   [x,y]  = (x /= y) && atS == y
latinDecompose     [x]  = True
latinDecompose      _   = False
```

7 Step 5: The runtime system

7.1 Commands: `CommandsLat.hs, latin.lexicon`

We do not want to recompile the whole system every time we add a new word. Actually, we may not even want to recompile at all — we may just want to be a user of a paradigm library. This is supported through the definition of an external lexicon.

We start by defining a set of commands. The commands of a morphology consists of a table that maps the name of a interface function with arguments to the real interface function.

```
commands =
 [
  ("d1puella",     ["rosa"],   app1 d1puella),
  ("d1puellaMasc", ["poeta"],  app1 d1puellaMasc),
  ("d2servus",     ["servus"], app1 d2servus),
  ("d2servusFem",  ["pinus"],  app1 d2servusFem),
  ("d2servusNeu",  ["virus"],  app1 d2servusNeu),
  ("d2bellum",     ["bellum"], app1 d2bellum),
  ("d2puer",       ["puer"],   app1 d2puer),
  ("d2liber",      ["liber"],  app1 d2liber),
  ("prep",         ["ad"],     app1 prep),
  ("v1amare",      ["amare"],  app1 v1amare),
  ("v2habere",     ["habere"], app1 v2habere)
  ]
```

When we have defined the set of commands, we can then start developing our external Latin lexicon (`latin.lexicon`):

```
   v1amare  amare
   v1amare  portare
   v1amare  demonstrare
   v1amare  laborare
```

7.2 The main runtime system

To get the runtime system up and running, we need to define a data type for our target language.

```
data Latin = Latin
 deriving Show
```

Next, we need to make our data type an instance of of the `Language` class. In this class, we define some functions needed for the runtime system. All definitions have a default definition, the internal dictionary can be empty, the list of commands may be empty and we may have no composite analysis in our morphology.

In the definition of the Latin morphology, our internal dictionary from `DictLat.hs` is called `latinDict`, our composite function, defined in `CompositeLat.hs`, is called `latinDecompose` and finally, our list of commands from `CommandsLat.hs` is added with the command `insertCommand` into a empty command structure `emptyC`.

```
instance Language Latin where
 internDict    _ = latinDict
 composition  _ = latinDecompose
 paradigms     _ = foldr insertCommand emptyC commands
```

The `main` function is simply to supply an object of a type that is an instance the `Language`, in this case the constructor `Latin`.

```
 main :: IO ()
 main = commonMain Latin
```

We have now developed our first fragment of a Latin morphology in FM!