Functional Morphology

Markus Forsberg and Aarne Ranta
Department of Computing Science
Chalmers University of Technology and the University of Gothenburg
{markus, aarne}@cs.chalmers.se

Abstract

This paper presents a methodology for implementing natural language morphology in the functional language Haskell. The main idea behind is simple: instead of working with untyped regular expressions, which is the state of the art of morphology in computational linguistics, we use finite functions over hereditarily finite algebraic datatypes. The definitions of these datatypes and functions are the language-dependent part of the morphology. The language-independent part consists of an untyped dictionary format which is used for synthesis of word forms, and a decorated trie, which is used for analysis.

Functional Morphology builds on ideas introduced by Huet in his computational linguistics toolkit Zen, which he has used to implement the morphology of Sanskrit. The goal has been to make it easy for linguists, who are not trained as functional programmers, to apply the ideas to new languages. As a proof of the productivity of the method, morphologies for Swedish, Italian, Russian, Spanish, and Latin have already been implemented using the library. The Latin morphology is used as a running example in this article.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming

General Terms

Languages, Design, Performance

Keywords

Morphological Description, Functional Programming, Linguistics, Embedded Languages, Finite Functions

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICFP'04, September 19–21, 2004, Snowbird, Utah, USA.
Copyright 2004 ACM 1-58113-905-5/04/0009 ...$5.00

1 Introduction

This paper presents a systematic way of developing natural language morphologies in a functional language. We think of functions and linguistic abstractions as strongly related in the sense that given a linguistic abstraction, it is, in most cases, natural and elegant to express it as a function. We feel that the methodology presented is yet another proof of this view.

An implementation of the methodology is presented, named *Functional Morphology* [8]. It can be viewed as an embedded domain-specific language in Haskell. Its basis are two type classes: `Param`, which formalizes the notion of a parameter type, and `Dict`, which formalizes the notion of a part of speech as represented in a dictionary. For these classes, Functional Morphology gives generic Haskell functions for morphological analysis and synthesis, as well as generation of code that presents the morphology in other formats, including Xerox Finite State Tools and relational databases.

The outline of the paper is the following: The morphology task is described in section 2, and then the contemporary approaches are discussed in section 3. The main part of the paper, section 4, focuses on describing the *Functional Morphology* library. We conclude in sections 5 and 6 with some results and a discussion.

2 Morphology

A morphology is a systematic description of words in a natural language. It describes a set of relations between words' *surface forms* and *lexical forms*. A word's surface form is its graphical or spoken form, and the lexical form is an analysis of the word into its *lemma* (also known as its *dictionary form*) and its *grammatical description*. This task is more precisely called *inflectional* morphology.

Yet another task, which is outside the scope of this paper, is *derivational* morphology, which describes how to construct new words in a language.

A clarifying example is the English word *functions'*. The graphical form *functions'* corresponds to the surface form of the word. A possible lexical form for the word *functions'* is *function +N +Pl +Gen*. From the analysis it can be read that the word can be analyzed into the lemma *function*, and the grammatical description *noun*, in *plural*, *genitive* case.

A morphological description has many applications, to mention a few: *machine translation*, *information retrieval*, *spelling and grammar checking* and *language learning*.

A morphology is a key component in machine translation, assuming that the aim is something more sophisticated than string to string translation. The grammatical properties of words are needed to handle linguistic phenomena such as *agreement*. Consider, for example, the subject-verb agreement in English — *Colorless green ideas **sleep** furiously*, not **Colorless green ideas **sleeps** furiously*.

In information retrieval, the use of a morphology is most easily explained through an example. Consider the case when you perform a search in a text for the word *car*. In the search result, you would also like to find information about *cars* and *car's*, but no information about *carts* and *careers*. A morphology is useful to do this kind of distinctions.

3 Implementations of Morphology

3.1 Finite State Technology

The main contemporary approach within computational morphology is finite state technology - the morphology is described with a regular expression [17, 26, 18, 15] that is compiled to a finite state transducer, using a finite state tool. Some of the tools available are: the commercial tool *XFST* [29], developed at Xerox, van Noord's [28] *finite state automata utilities*, AT&T's [19] *FSM library* and Forsberg's [6] *FST Studio*.

Finite state technology is a popular choice since finite state transducers provide a compact representation of the implemented morphology, and the lookup time is close to constant in the size of the lexicon.

Finite state technology is based on the notion of a *regular relation*. A regular relation is a set of n-tuples of words. Regular languages are a special case, with $n = 1$. Morphology tools such as XFST work with 2-place relations. They come with an extended regular expression notation for easy manipulation of symbol and word pairs. Such expressions are compiled into *finite-state transducers*, which are like finite-state automata, but their arcs are labelled by pairs of symbols rather than just symbols. Strings consisting of the first components of these pairs are called the *upper language* of the transducer, and strings consisting of the second components are called the *lower language*. A transducer is typically used so that the upper language contains structural descriptions of word forms and the lower language contains the forms themselves.

A trivial example of a regular relation is the description of the inflection of three English nouns in number. The code is XFST source code, where the `|` is the union operator, and `.x.` is the cross product of the strings. Concatenation is expressed by juxtaposition.

```
NOUN = "table" | "horse" | "cat"

INFL = NOUN .x. "Sg" | NOUN "s" .x. "Pl"
```

If a transducer is compiled from this regular relation, and applied *upward* with the string `"tables"`, it will return {`"Pl"`}. If the built transducer is applied *downward* with the string `"Sg"`, it will return {`"table", "horse", "cat"`}.

One problem with finite-state tranducers is that cycles (corresponding to Kleene stars in regular expressions), can appear anywhere in them. This increases the complexity of compilation so that it can be exponential. Compiling a morphological description to a transducer has been reported to last several days, and sometimes small changes in the source code can make a huge difference. Another problem is that transducers cannot generally be made deterministic for sequences of symbols (they are of course deterministic for sequences of symbol pairs). This means that analysis and synthesis can be worse than linear in the size of the input.

3.2 The Zen Linguistic Toolkit

Huet has used the functional language Caml to build a Sanskrit dictionary and morphological analysis and synthesis. [12]. He has generalized the ideas used for Sanskrit to a toolkit for computational linguistics, Zen [13]. The key idea is to exploit the expressive power of a functional language to define a morphology on a high level, higher than regular expressions. Such definitions are moreover safe, in the sense that the type checker guarantees that all words are defined correctly as required by the definitions of different parts of speech.

The analysis of words in Zen is performed by using *tries*. A trie is a special case of a finite-state automaton, which has no cycles. As Huet points out, the extra power added by cycles is not needed for the morphological description inside words, but, at most, between words. This extra power is needed in languages like Sanskrit where word boundaries are not visible and adjacent words can affect each other (this phenomenon is known as *sandhi*). It is also needed in languages like Swedish where compound words can be formed almost *ad libitum*, and words often have special forms used in compounds. Compositions of tries, with cycles possible only on word boundaries, have a much nicer computational behaviour than full-scale transducers.

3.3 Grammatical Framework

The Grammatical Framework GF [25] is a special-purpose functional language for defining grammars, including ones for natural languages. One part of a grammar is a morphology, and therefore GF has to be capable of defining morphology. In a sense, this is trivial, since morphology requires strictly less expressive power than syntax (regular languages as opposed to context-free languages and beyond). At the same time, using a grammar formalism for morphology is overkill, and may result in severely suboptimal implementations.

One way to see the Functional Morphology library described in this paper is as a fragment of GF embedded in Haskell. The `Param` and `Dict` classes correspond to constructs that are hard-wired in GF: *parameter types* and *linearization types*, respectively. Given this close correspondence, it is no wonder that it is very easy to generate GF code from a Functional Morphology description. On the other hand, the way morphological analysis is implemented efficiently using tries has later been adopted in GF, so that the argument on efficiency is no longer so important. Thus one can see the morphology fragment of GF as an instance of the methodology of Functional Morphology. However, complicated morphological rules (such as stem-internal vowel changes) are easier to write in Haskell than in GF, since Haskell provides more powerful list and string processing than GF.

4 Functional morphology

4.1 Background

The goal of our work is to provide a freely available open-source library that provides a high level of abstraction for defining natural language morphologies. The examples used in this article are collected from Latin morphology. Our Latin morphology is based on the descriptions provided by [20, 5, 16, 2].

Our work is heavily influenced by Huet's functional description of Sanskrit [12] and his Zen Toolkit [13]. The analyzer provided by Functional Morphology can be seen as a Haskell version of Huet's "reference implementation" in Caml. At the same time, we aim to provide a language-independent high-level *front-end* to those tools that makes it possible to define a morphology with modest training in functional programming.

The idea of using an embedded language with a support for code generation is related to Claessen's hardware description language Lava [4], which is compiled into VHDL. For the same reasons as it is important for Lava to generate VHDL—the needs of the main stream community—we generate regular expressions in the XFST and LEXC formats.

Functional Morphology is based on an old idea, which has been around for over 2000 years, that of *inflection tables*. An inflection table captures an inflectional regularity in a language. A morphology is a set of tables and a dictionary. A dictionary consists of lemmas, or dictionary forms, tagged with pointers to tables.

An inflection table displaying the inflection of regular nouns in English, illustrated with the lemma *function*, is shown below.

	Case	
Number	Nominative	Genitive
Singular	*function*	*function's*
Plural	*functions*	*functions'*

Different ways of describing morphologies were identified by Hockett [9] in 1950's. The view of a morphology as a set of inflection tables he calls *word and paradigm*. The paradigm is an inflection table, and the word is an example word that represent a group of words with the same inflection table.

In a sense, the research problem of describing inflectional morphologies is already solved: how to fully describe a language's inflectional morphology in the languages we studied is already known. But the are still problematic issues which are related to the size of a typical morphology. A morphology covering a dictionary of a language, if written out in full form lexicon format, can be as large as 1-10 million words, each tagged with their grammatical description.

The size of the morphology demands two things: first, we need an efficient way of describing the words in the morphology, generalize as much as possible to minimize the effort of implementing the morphology, and secondly, we need a compact representation of the morphology that has an efficient lookup function.

4.2 Methodology

The methodology suggests that paradigms, i.e. inflection tables, should be defined as finite functions over an enumerable, hereditarily finite, algebraic data type describing the parameters of the paradigm. These functions are later translated to a *dictionary*, which is a language-independent datastructure designed to support analyzers, synthesizers, and generation of code in other formats than Haskell.

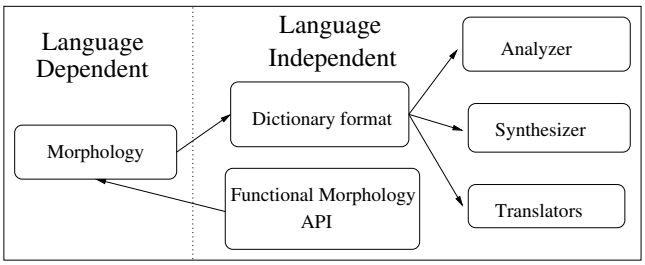

Figure 1. Functional Morphology system overview

All parameter types are instances of the `Param` class, which is an extension of the built-in `Enum` and `Bounded` class, to be able to define enumerable, finite types over hierarchical data types.

Parts of speech are modelled by instances of the class `Dict`, which automate the translation from a paradigm to the `Dictionary` type.

4.3 System overview

A Functional Morphology system consists of two parts, one language dependent part, and one language independent part, illustrated in figure 1.

The language dependent part is what the morphology implementor has to provide, and it consists of a *type system*, an *inflection engine* and a *dictionary*. The type system gives all word classes and their inflection and inherent parameters, and instances of the `Param` class and the `Dict` class. The inflection machinery defines all valid inflection tables, i.e. all *paradigms*, as finite functions. The dictionary lists all words in dictionary form with its paradigm in the language.

Defining the type system and the inflection machinery can be a demanding task, where you not only need to be knowledgeable about the language in question, but also have to have some understanding about functional programming. The libraries provided by Functional Morphology simplifies this step.

However, when the general framework has been defined, which is actually a new library built on top of ours, it is easy for a lexicographer to add new words, and this can be done with limited or no knowledge about functional programming. The lexicographer does not even have to be knowledgeable about the inner workings of a morphology, it is sufficient that she knows the inflectional patterns of words in the target language.

4.4 Technical details

4.4.1 Parameter types

In grammars, words are divided into classes according to 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 `String`. The choice of `String` as a return type will be problematized in section 4.4.5 and another type, `Str`, will be introduced.

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

More generally, a finite function in Functional Morphology, is a function `f` from a parameter type `P` to strings.

```
f :: P -> String
```

Note that the finite functions have a single argument. This is, however, not a limitation, because we can construct arbitrarily complex single types with tuple-like constructors.

4.4.2 Type hierarchy

A naive way of describing a class of words is by using the cross product of all parameters. This would in many languages lead to a serious over-generation of cases that do not exist in the language.

An example is the Latin verbs, where the cross product of the inflection parameters generates 1260 forms (three persons, two numbers, six tenses, seven moods and five cases[1]), but only 147 forms actually exist, which is just about a ninth of 1260.

This problem is easily avoided in a language like Haskell that has algebraic data types, where data types are not only enumerated, but also complex types with constructors that have type parameters as arguments.

The type system for Latin verbs can be defined with the data types below, that exactly describes the 147 forms that exist in Latin verb conjugation:

```
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                           |
```

[1]The verb inflection in case only appears in the gerund and supine mood, and only some of the six cases are possible.

```
  GerundAcc                          |
  GerundAbl                          |
  SupineAcc                          |
  SupineAblative
```

This representation gives a correct description of what forms exist, and it is hence linguistically more satisfying than a cross-product of features. The type system moreover enables a completeness check to be performed.

4.4.3 Tables and finite functions

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.

To illustrate the convenience with using finite functions instead of tables, consider the inflection table of the Latin word *rosa* (Eng. *rose*):

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

The word has two inflection parameters, case and number, that, as discussed in section 4.4.1, can be described in Haskell with algebraic data types.

```
data Case     = Nominative | Vocative |
                Accusative | Genitive |
                Dative     | Ablative
data Number   = Singular   | Plural
data NounForm = NounForm Case Number
```

The inflection table can be viewed as a list of pairs, where the first component of a pair is an inflection parameter, and the second component is an inflected word. The inflection table of *rosa* is described, in the definition of `rosa` below, as a list of pairs.

```
rosa :: [(NounForm,String)]
rosa =
 [
  (NounForm Singular Nominative,"rosa"),
  (NounForm Singular Vocative,"rosa"),
  (NounForm Singular Accusative,"rosam"),
  (NounForm Singular Genitive,"rosae"),
  (NounForm Singular Dative,"rosae"),
  (NounForm Singular Ablative,"rosa"),
  (NounForm Plural Nominative,"rosae"),
  (NounForm Plural Vocative,"rosae"),
  (NounForm Plural Accusative,"rosas"),
  (NounForm Plural Genitive,"rosarum"),
  (NounForm Plural Dative,"rosis"),
  (NounForm Plural Ablative,"rosis")
 ]
```

The type `NounForm` is finite, so instead of writing these kinds of tables, we can write a finite function that describes this table more compactly. We could even go a step further, and first define a function that describes all nouns that inflects in the same way as the noun *rosa*, i.e. defining a paradigm.

```
rosaParadigm :: String → Noun
rosaParadigm rosa (NounForm n c) =
 let rosae = rosa ++ "e"
     rosis = init rosa ++ "is"
  in
   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"
                  _          → rosis
```

It may seem that not much has been gained, except that the twelve cases have been collapsed to nine, and we have achieved some sharing of `rosa` and `rosae`.

However, the gain is clearer when defining the paradigm for *dea* (Eng. *goddess*), that inflects in the same way, with the exception of two case, plural dative and ablative.

```
dea :: Noun
dea nf =
  case nf of
   NounForm Plural Dative   → dea
   NounForm Plural Ablative → dea
   _                        → rosaParadigm dea nf
 where dea = "dea"
```

Given the paradigm of *rosa*, `rosaParadigm`, we can describe the inflection tables of other nouns in the same paradigm, such as *causa* (Eng. *cause*) and *barba* (Eng. *beard*).

```
rosa, causa, barba :: Noun
rosa  = rosaParadigm "rosa"
causa = rosaParadigm "causa"
barba = rosaParadigm "barba"
```

4.4.4 Turning a function into a table

The most important function of Functional Morphology is `table`, that translates a finite function into a list of pairs. This is done by ensuring that the parameter type is of the `Param` class, which enables us to generate all forms with the class function `values`.

```
table :: Param a ⇒ (a → Str) → [(a,Str)]
table f = [(v, f v) | v ← values]
```

A function would only be good for generating forms, but with `table`, the function can be compiled into lookup tables and further to tries to perform analysis as well.

4.4.5 String values

The use of a single string for representing a word is too restricted, because words can have *free variation*, i.e., that two or more words have the same morphological meaning, but are spelled differently. Yet another exception is *missing forms*, some inflection tables may have missing cases.

Free variation exists in the Latin noun *domus* (Eng. *home*) in singular dative, *domui* or *domo*, in plural accusative, *domus* or *domos*, and in plural genitive, *domuum* or *domorum*.

Missing forms appear in the Latin noun *vis* (Eng. *violence, force*), a noun that is *defective* in linguistic terms.

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

These two observations lead us to represent a word with the abstract type `Str`, which is simply a list of strings. The empty list corresponds to the missing case.

```
type Str = [String]
```

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 = []
```

The inflection table of *vis* can be described with the function `vis` below.

```
vis :: Noun
vis (NounForm n c) =
  case n of
   Singular → case c of
                Nominative → mkStr $ vi ++ "s"
                Accusative → mkStr $ vi ++ "m"
                Ablative   → mkStr vi
                _          → nonExist
   Plural   → mkStr $
              case c of
                Genitive   → vir ++ "ium"
                Dative     → viribus
                Ablative   → viribus
                _          → vir ++ "es"
 where vi = "vi"
       vir = vi ++ "r"
       viribus = vir ++ "ibus"
```

4.4.6 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.[2] 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.

4.4.7 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)
excepts f es p = maybe (f p) id $ lookup p es
```

The paradigm of *dea* defined in section 4.4.3 can be described with the function `dea` using the exception `excepts`.

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

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)
missing f as = excepts f [(a,nonExist) | a ← as]

only :: Param a ⇒ (a → Str) → [a] → (a → Str)
only f as = missing f [a | a ← values, notElem a as]
```

The paradigm of *vis* described in section 4.4.5, can be described with the `only` exception and the paradigm of *hostis* (Eng. enemy).

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

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.

[2] E.g. in Arabic, triples of consonants are a natural way to represent the so-called roots of words.

```
variants :: Param a ⇒ (a → Str) → [(a,String)] →
                                   (a → Str)
variants f es p =
  maybe (f p) (reverse . (: f p)) $ lookup p es
```

4.4.8 Dictionary

The `Dictionary` type is the core of Functional Morphology, in the sense that the morphology description denotes a `Dictionary`. The `Dictionary` is a language-independent representation of a morphology, that is chosen to make generation to other formats easy.

A `Dictionary` is a list of `Entry`, where an `Entry` corresponds to a specific dictionary word.

```
type Dictionary = [Entry]
```

An `Entry` consists of the dictionary word, the part of speech (category) symbol, a list of the inherent parameters, and the word's, lacking a better word, untyped inflection table.

```
type Dictionary       = [Entry]
type Entry = (Dictionary_Word, Category,
              [Inherent], Inflection_Table)
type Dictionary_Word  = String
type Category         = String
type Inherent         = String
type Parameter        = String
type Inflection_Table = [(Parameter,(Attr,Str))]
```

The `Attr` type and definitions containing this type concerns the handling of composite forms, that will be explained later in section 4.6.

To be able to generate the `Dictionary` type automatically, a class `Dict` has been defined. Only composite types, describing the inflection parameters of a part of speech, should normally be an instance of the `Dict` class.

```
class Param a ⇒ Dict a where
  dictword :: (a → Str) → String
  category :: (a → Str) → String
  defaultAttr :: (a → Str) → Attr
  attrException :: (a → Str) → [(a,Attr)]
  dictword f = concat $ take 1 $ f value0
  category = const "Undefined"
  defaultAttr = const atW
  attrException = const []
```

Note that all class functions have a default definition, but usually we have to at least give a definition of `category`, that gives the name of the part of speech of a particular parameter type. It's impossible to give a reasonable default definition of `category`; it would require that we have types as first class objects.

It may be surprising that `category` and `defaultAttr` are higher-order functions. This is simply a type hack that forces the inference of the correct class instance without the need to provide an object of the type. Normally, the function argument is an inflection table (cf. the definition of `entryI` below).

The most important function defined for types in `Dict` is `entryI`, which, given a paradigm and a list of inherent features, creates an `Entry`. However, most categories lack inherent features, so the function `entry` is used in most cases, with an empty list of inherent features.

```
entryI :: Dict a ⇒ (a → Str) → [Inherent] → Entry
entryI f ihs = (dictword f, category f, ihs, infTable f)

entry  :: Dict a ⇒ (a → Str) → Entry
entry f = entryI f []
```

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
```

Finally, 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`.

```
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]]
  ]
```

The dictionary above consists of 11 dictionary entries, which defines a lexicon of 132 full form words. Note that when using exceptions, the use of interface functions has to be postponed. We could define exceptions on the entry level, but we would then loose the type safety.

Even more productive are the interface functions for Latin verbs. Consider the dictionary `latinVerbs` below, that uses the interface functions `v1amare` (Eng. *to love*) and `v2habere` (Eng. *to have*).

```
latinVerbs :: Dictionary
latinVerbs =
 dictionary $
  [
   v1amare  "amare",
   v1amare  "portare",
   v1amare  "demonstrare",
   v1amare  "laborare",
   v2habere "monere",
   v2habere "admonere",
   v2habere "habere"
   ]
```

The dictionary `latinVerbs` consists of 7 dictionary entries, that defines a lexicon of as many as 1029 full form words.

4.4.9 External dictionary

When a set of interface functions have been defined, we don't want to recompile the system every time we add a new regular word. Instead, we define an external dictionary format, with a translation function to the internal `Dictionary`. The syntax of the external dictionary format is straightforward: just a listing of the words with their paradigms. The first entries of the dictionary `latinVerbs` are written

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

Notice that the external dictionary format is a very simple special-purpose language implemented on top of the morphology of one language. This is the only language that a person extending a lexicon needs to learn.

4.4.10 Code generation

The `Dictionary` format, described in section 4.4.8, has been defined with generation in mind. It is usually easy to define a translation to another format. Let us look at an example of how the *LEXC* source code is generated. The size of the function `prLEXC`, not the details, is the interesting part. It is just 18 lines. The functions not defined in the function, is part of Haskell's standard Prelude or the standard API of Functional Morphology.

```
prLEXC :: Dictionary → String
prLEXC = unlines . (["LEXICON Root",[]] ++) . (++ ["END",[]]) .
                   map (uncurry prLEXCRules) . classifyDict

prLEXCRules :: Ident → [Entry] → String
prLEXCRules cat entries =
 unlines $ [[],"! category " ++ cat,[]] ++
             (map (prEntry . noAttr) entries)
  where
    prEntry (stem,_,inhs,tbl) =
     concat (map (prForm stem inhs) (existingForms tbl))
    prForm stem inhs (a,b) =
     unlines
      [x ++ ":" ++ stem ++ prTags (a:inhs) ++ " # ;" | x <- b]
    prTags ts =
     concat
      ["+" ++ w | t <- ts, w <- words (prFlat t)]
    altsLEXC cs =
     unwords $ intersperse " # ;" [ s  | s <- cs]
```

Currently, the following formats are supported by Functional Morphology.

Full form lexicon .A full form lexicon is a listing of all word forms with their analyses, in alphabetical order, in the lexicon.

Inflection tables. Printer-quality tables typeset in LaTeX

GF grammar source code. Translation to a Grammatical Framework grammar.

XML. An XML[27] representation of the morphological lexicon.

XFST source code. Source code for a simple, non-cyclic transducer in the Xerox notation.

LEXC source code. Source code for LEXC format, a version of XFST that is optimized for morphological descriptions.

Relational database. A database described with SQL source code.

Decorated tries. An analyzer for the morphology as a decorated trie.

CGI. A web server for querying and updating the morphological lexicon.[3]

4.5 Trie analyzer

The analyzer is a key component in a morphology system — to analyze a word into its lemma and its grammatical description. Synthesizers are also interesting, that is, given an analysis, produce the word form. In a trivial sense, an analyzer already exists through the XFST/LEXC formats, but Functional Morphology also provides its own analyzer.

Decorated tries is currently used instead of transducers for analysis in our implementation. Decorated tries can be considered as a specialized version of one of the languages in a transducer, that is deterministic with respect to that language, hence prefix-minimal. If we have an undecorated trie, we can also achieve total minimality by sharing, as described by Huet [14]; full-scale transducers can even achieve suffix sharing by using decorated edges. This approach has been used by Huet [12], when defining a morphology for Sanskrit. The trie is size-optimized by using a symbol table for the return value (the grammatical description).

4.6 Composite forms

Some natural languages have compound words — words composed from other words. A typical example is the (outdated) German word for a computer, *Datenverarbeitungsanlage*, composed from *Daten*, *Verarbeitung*, and *Anlage*. If such words are uncommon, they can be put to the lexicon, but if they are a core feature of the language (as in German), this productivity must be described in the morphology. Highly inspired by Huet's glue function [14], we have solved the problem by tagging all words with a special type `Attr` that is just a code for how a word can be combined with other words. At the analysis phase, the trie is iterated, and words are decomposed according to these parameters.

The `Attr` type is simply a integer. Together with a set of constants `atW`, `atP`, `atWP` and `atS`, we can describe how a word can be combined with another. The `atW` for stand-alone words, `atP` for words that can only be a prefix of other words, `atWP` for words that can be a stand-alone word and a prefix, and finally, `atS`, for words that can only be a suffix of other words.

```
type Attr = Int

atW, atP, atWP, atS :: Attr
(atW, atP, atWP, atS) = (0,1,2,3)
```

As an example, we will describe how to add the productive question particle *ne* in Latin, that can be added as a suffix to any word in Latin, and has the interrogative meaning of a questioning the word.

We begin by defining a type for the particle, and instantiate it in `Param`. The `Invariant` type expresses that the particle is not inflected.

```
data ParticleForm = ParticleForm Invariant
 deriving (Show,Eq)

type Particle    = ParticleForm  -> Str

instance Param ParticleForm where
    values     = [ParticleForm p | p <- values]
    prValue _  = "Invariant"
```

We continue by instantiating `ParticleForm` in `Dict`, where we also give a definition for `defaultAttr` with `atS`, that expresses that the words of this form can only appear as a suffix to another word, not as a word on its own.

```
instance Dict ParticleForm
    where category    _ = "Particle"
          defaultAttr _ = atS
```

We then define an interface function `particle` and add *ne* to our dictionary.

```
makeParticle :: String -> Particle
makeParticle s _ = mkStr s

particle :: String -> Entry
particle = entry . makeParticle

dictLat :: Dictionary
dictLat = dictionary $
          [
          ...
          particle "ne"
          ]
```

Analyzing the word *servumne*, the questioning that the object in a phrase is a slave or a servant, gives the following analysis in Functional Morphology:

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

5 Results

The following morphologies have been implemented in Functional Morphology: a Swedish inflection machinery and a lexicon of 15,000 words [8]; a Spanish inflection machinery + lexicon of 10,000 words [1]; major parts of the inflection machinery + lexicon

[3] In a previous version, a CGI morphology web server was generated. Meijer's [21] CGI library was used, further modified by Panne. There exists a prototype web server [7] for Swedish. However, the CGI implementation scaled up poorly, so it is no longer generated. This is to be replaced by a SQL database and PHP.

for Russian [3], Italian [24], and Latin [8]. Comprehensive inflection engines for Finnish and French have been written following the same method but using GF as source language [23]

One interesting fact is that the Master students had very limited knowledge of Haskell before they started their projects, but still managed to produce competitive morphology implementations.

An interface between morphology and syntax, through the Grammatical Framework, exists. An implemented morphology can directly be used as a resource for a grammatical description.

The analyzer tags words with a speed of 2k-50k words/second (depending on how much compound analysis is involved), a speed that compares with finite state transducers. The analyzer is often compiled faster than XFST's finite state transducers, because Kleene's star is disallowed within a word description.

6 Discussion

One way of viewing Functional Morphology is as a domain specific embedded language [10, 11], with the functional programming language Haskell [22] as host language.

There are a lot of features that make Haskell suitable as a host language, to mention a few: *a strong type system*, *polymorphism*, *class system*, and *higher-order functions*. Functional Morphology uses all of the mentioned features.

One could wonder if the power and freedom provided by a general-purpose programming language does not lead to problems, in terms of errors and inconsistency. Functional Morphology avoids this by requiring from the user that the definition denotes an object of a given type, i.e. the user has full freedom to use the whole power of Haskell as long as she respects the type system of Functional Morphology.

Embedding a language into another may also lead to efficiency issues — an embedded language cannot usually compete with a DSL that has been optimized for the given problem domain. This is avoided in Functional Morphology by generating other formats which provide the efficiency needed.

A simple representation of the morphology in the core system has been chosen, which enables easy generation of other formats. This approach makes the framework more easily adaptable to future applications, which may require new formats. It also enforces the *single-source* idea, i.e. a general single format is used that generates the formats of interest. A single source solves the problems of maintainability and inconsistency.

Programming constructs and features available in a functional framework make it is easier to capture generalizations that may even transcend over different languages. It is no coincidence that Spanish, French and Italian are among the languages we have implemented: the languages' morphology are relatively close, so some of the type systems and function definitions could be reused.

We believe that we provide a higher level of abstraction than the mainstream tools using regular relations, which results in a faster development and easier adaption. Not only does the morphology implementor have a nice and flexible framework to work within, but she gets a lot for free through the translators, and will also profit from further development of the system.

7 References

[1] I. Andersson and T. Söderberg. Spanish Morphology – implemented in a functional programming language. Master's Thesis in Computational Linguistics, 2003. `http://www.cling.gu.se/theses/finished.html`.

[2] C. E. Bennett. *A Latin Grammar*. Allyn and Bacon, Boston and Chicago, 1913.

[3] L. Bogavac. Functional Morphology for Russian. Master's Thesis in Computing Science, 2004.

[4] K. Claessen. *An Embedded Language Approach to Hardware Description and Verification*. PhD thesis, Chalmers University of Technology, 2000.

[5] E. Conrad. Latin grammar. `www.math.ohio-state.edu/~econrad/lang/latin.html`, 2004.

[6] M. Forsberg. Fststudio. `http://www.cs.chalmers.se/~markus/fstStudio`

[7] M. Forsberg and A. Ranta. Svenska ord. `http://www.cs.chalmers.se/~markus/svenska`, 2002.

[8] M. Forsberg and A. Ranta. Functional morphology. `http://www.cs.chalmers.se/~markus/FM`, 2004.

[9] C. F. Hockett. Two models of grammatical description. *Word*, 10:210–234, 1954.

[10] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4), 1996.

[11] P. Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[12] G. Huet. Sanskrit site. Program and documentation, `http://pauillac.inria.fr/~huet/SKT/`, 2000.

[13] G. Huet. The Zen Computational Linguistics Toolkit. `http://pauillac.inria.fr/~huet/`, 2002.

[14] G. Huet. Transducers as lexicon morphisms, phonemic segmentation by euphony analysis, application to a sanskrit tagger. *Available:* `http://pauillac.inria.fr/~huet/FREE/`, 2003.

[15] L. K. Kenneth R. Beesley. *Finite State Morphology*. CSLI Publications, United States, 2003.

[16] G. Klyve. *Latin Grammar*. Hodder & Stoughton Ltd., London, 2002.

[17] K. Koskenniemi. *Two-level morphology: a general computational model for word-form recognition and production*. PhD thesis, University of Helsinki, 1983.

[18] G. G. L. Karttunen, J-P Chanond and A. Schille. Regular expressions for language engineering. *Natural Language Engineering*, 2:305–328, 1996.

[19] A. Labs-Research. At&t fsm library. http://www.research.att.com/sw/tools/fsm/.

[20] J. Lambek. A mathematician looks at the latin conjugation. *Theoretical Linguistics*, 1977.

[21] E. Meijer and J. van Dijk. Perl for swine: Cgi programming in haskell. *Proc. First Workshop on Functional Programming*, 1996.

[22] S. Peyton Jones and J. Hughes. Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language. Available from http://www.haskell.org, February 1999.

[23] A. Ranta. Grammatical Framework Homepage, 2000–2004. www.cs.chalmers.se/~aarne/GF/.

[24] A. Ranta. 1+n representations of Italian morphology. Essays dedicated to Jan von Plato on the occasion of his 50th birthday, http://www.valt.helsinki.fi/kfil/jvp50.htm, 2001.

[25] A. Ranta. Grammatical Framework: A Type-theoretical Grammar Formalism. *The Journal of Functional Programming*, 14(2):145–189, 2004.

[26] M. K. Ronald M. Kaplan. Regular Models of Phonological Rule Systems. *Computational lingustics*, pages 331–380, 1994.

[27] The World Wide Web Consortium. Extensible Markup Language (XML). http://www.w3.org/XML/, 2000.

[28] G. van Noord. Finite state automata utilities. http://odur.let.rug.nl/~vannoord/Fsa/

[29] Xerox. Xerox finite-state compiler. http://www.xrce.xerox.com/competencies /content-analysis/fsCompiler/.