ElixirFM — Implementation of Functional Arabic Morphology

Otakar Smrž
Institute of Formal and Applied Linguistics
Faculty of Mathematics and Physics
Charles University in Prague
otakar.smrz@mff.cuni.cz

Abstract

Functional Arabic Morphology is a formulation of the Arabic inflectional system seeking the working interface between morphology and syntax. ElixirFM is its high-level implementation that reuses and extends the Functional Morphology library for Haskell. Inflection and derivation are modeled in terms of paradigms, grammatical categories, lexemes and word classes. The computation of analysis or generation is conceptually distinguished from the general-purpose linguistic model. The lexicon of ElixirFM is designed with respect to abstraction, yet is no more complicated than printed dictionaries. It is derived from the open-source Buckwalter lexicon and is enhanced with information sourcing from the syntactic annotations of the Prague Arabic Dependency Treebank.

1 Overview

One can observe several different streams both in the computational and the purely linguistic modeling of morphology. Some are motivated by the need to analyze word forms as to their compositional structure, others consider word inflection as being driven by the underlying system of the language and the formal requirements of its grammar.

In Section 2, before we focus on the principles of ElixirFM, we briefly follow the characterization of morphological theories presented by Stump (2001) and extend the classification to the most prominent computational models of Arabic morphology (Beesley, 2001; Buckwalter, 2002; Habash et al., 2005; El Dada and Ranta, 2006).

In Section 3, we survey some of the categories of the syntax–morphology interface in Modern Written Arabic, as described by the Functional Arabic Morphology. In passing, we will introduce the basic concepts of programming in Haskell, a modern purely functional language that is an excellent choice for declarative generative modeling of morphologies, as Forsberg and Ranta (2004) have shown.

Section 4 will be devoted to describing the lexicon of ElixirFM. We will develop a so-called domain-specific language embedded in Haskell with which we will achieve lexical definitions that are simultaneously a source code that can be checked for consistency, a data structure ready for rather independent processing, and still an easy-to-read-and-edit document resembling the printed dictionaries.

In Section 5, we will illustrate how rules of inflection and derivation interact with the parameters of the grammar and the lexical information. We will demonstrate, also with reference to the Functional Morphology library (Forsberg and Ranta, 2004), the reusability of the system in many applications, including computational analysis and generation in various modes, exploring and exporting of the lexicon, printing of the inflectional paradigms, etc.

2 Morphological Models

According to Stump (2001), morphological theories can be classified along two scales. The first one deals with the core or the process of inflection:

lexical theories associate word's morphosyntactic properties with *affixes*

inferential theories consider inflection as a result of operations on *lexemes*; morphosyntactic prop-

Proceedings of the 5th Workshop on Important Unresolved Matters, pages 1–8,
Prague, Czech Republic, June 2007.

erties are expressed by the *rules* that relate the form in a given paradigm to the lexeme

The second opposition concerns the question of inferability of meaning, and theories divide into:

incremental words *acquire* morphosyntactic properties only in connection with acquiring the inflectional exponents of those properties

realizational association of a set of properties with a word *licenses* the introduction of the exponents into the word's morphology

Evidence favoring inferential–realizational theories over the other three approaches is presented by Stump (2001) as well as Baerman et al. (2006) or Spencer (2004). In trying to classify the implementations of Arabic morphological models, let us reconsider this cross-linguistic observation:

> The morphosyntactic properties associated with an inflected word's individual inflectional markings may underdetermine the properties associated with the word as a whole. (Stump, 2001, p. 7)

How do the current morphological analyzers interpret, for instance, the number and gender of the Arabic broken masculine plurals *ǧudud* جُدُد *new ones* or *quḍāh* قُضَاة *judges*, or the case of *mustawan* مُستَوًى *a level*? Do they identify the values of these features that the syntax actually operates with, or is the resolution hindered by some too generic assumptions about the relation between meaning and form?

Many of the computational models of Arabic morphology, including in particular (Beesley, 2001), (Ramsay and Mansur, 2001) or (Buckwalter, 2002), are *lexical* in nature. As they are not designed in connection with any syntax–morphology interface, their interpretations are destined to be *incremental*.

Some signs of a *lexical–realizational* system can be found in (Habash, 2004). The author mentions and fixes the problem of underdetermination of inherent number with broken plurals, when developing a generative counterpart to (Buckwalter, 2002).

The computational models in (Soudi et al., 2001) and (Habash et al., 2005) attempt the *inferential–realizational* direction. Unfortunately, they implement only sections of the Arabic morphological system. The Arabic resource grammar in the Grammatical Framework (El Dada and Ranta, 2006) is perhaps the most complete inferential–realizational implementation to date. Its style is compatible with the linguistic description in e.g. (Fischer, 2001) or (Badawi et al., 2004), but the lexicon is now very limited and some other extensions for data-oriented computational applications are still needed.

ElixirFM is inspired by the methodology in (Forsberg and Ranta, 2004) and by functional programming, just like the Arabic GF is (El Dada and Ranta, 2006). Nonetheless, ElixirFM reuses the Buckwalter lexicon (2002) and the annotations in the Prague Arabic Dependency Treebank (Hajič et al., 2004), and implements yet more refined linguistic model.

3 Morphosyntactic Categories

Functional Arabic Morphology and ElixirFM reestablish the system of *inflectional* and *inherent* morphosyntactic properties (alternatively named grammatical categories or features) and distinguish precisely the senses of their use in the grammar.

In Haskell, all these categories can be represented as distinct data types that consist of uniquely identified values. We can for instance declare that the category of case in Arabic discerns three values, that we also distinguish three values for number or person, or two values of the given names for verbal voice:

```
data Case = Nominative | Genitive |
            Accusative
data Number = Singular | Dual | Plural
data Person = First | Second | Third
data Voice = Active | Passive
```

All these declarations introduce new enumerated types, and we can use some easily-defined methods of Haskell to work with them. If we load this (slightly extended) program into the interpreter,[1] we can e.g. ask what category the value `Genitive` belongs to (seen as the `::` type signature), or have it evaluate the list of the values that `Person` allows:

```
? :type Genitive    → Genitive :: Case
? enum :: [Person] → [First,Second,Third]
```

Lists in Haskell are data types that can be parametrized by the type that they contain. So, the value `[Active, Active, Passive]` is a list of three elements of type `Voice`, and we can write this if necessary as the signature `:: [Voice]`. Lists can also

[1] `http://www.haskell.org/`

be empty or have just one single element. We denote lists containing some type `a` as being of type `[a]`.

Haskell provides a number of useful types already, such as the enumerated boolean type or the parametric type for working with optional values:

```
data Bool = True | False
data Maybe a = Just a | Nothing
```

Similarly, we can define a type that couples other values together. In the general form, we can write

```
data Couple a b = a :-: b
```

which introduces the value `:-:` as a container for some value of type `a` and another of type `b`.[2]

Let us return to the grammatical categories. Inflection of nominals is subject to several formal requirements, which different morphological models decompose differently into features and values that are not always complete with respect to the inflectional system, nor mutually orthogonal. We will explain what we mean by revisiting the notions of state and definiteness in contemporary written Arabic.

To minimize the confusion of terms, we will depart from the formulation presented in (El Dada and Ranta, 2006). In there, there is only one relevant category, which we can reimplement as `State'`:

```
data State' = Def | Indef | Const
```

Variation of the values of `State'` would enable generating the forms *al-kitābu* الكِتَابُ def., *kitābun* كِتَابٌ indef., and *kitābu* كِتَابُ const. for the nominative singular of *book*. This seems fine until we explore more inflectional classes. The very variation for the nominative plural masculine of the adjective *high* gets *ar-rafīūna* الرَّفِيعُونَ def., *rafīūna* رَفِيعُونَ indef., and *rafīū* رَفِيعُو const. But what value does the form *ar-rafīū* الرَّفِيعُو, found in improper annexations such as in *al-masūlūna 'r-rafīū 'l-mustawā* المَسؤُولُونَ الرَّفِيعُو المُستَوَى *the-officials the-highs-of the-level*, receive?

It is interesting to consult for instance (Fischer, 2001), where state has exactly the values of `State'`, but where the definite state `Def` covers even forms without the prefixed *al-* ال article, since also some separate words like *lā* لَا *no* or *yā* يَا *oh* can have the effects on inflection that the definite article has. To distinguish all the forms, we might think of keeping state in the sense of Fischer, and adding a boolean feature for the presence of the definite article... However, we would get one unacceptable combination of the values claiming the presence of the definite article and yet the indefinite state, i.e. possibly the indefinite article or the diptotic declension.

Functional Arabic Morphology refactors the six different kinds of forms (if we consider all inflectional situations) depending on two parameters. The first controls prefixation of the (virtual) definite article, the other reduces some suffixes if the word is a head of an annexation. In ElixirFM, we define these parameters as type synonyms to what we recall:

```
type Definite = Maybe Bool
type Annexing = Bool
```

The `Definite` values include `Just True` for forms with the definite article, `Just False` for forms in some compounds or after *lā* لَا or *yā* يَا (absolute negatives or vocatives), and `Nothing` for forms that reject the definite article for other reasons.

Functional Arabic Morphology considers state as a result of coupling the two independent parameters:

```
type State = Couple Definite Annexing
```

Thus, the indefinite state `Indef` describes a word void of the definite article(s) and not heading an annexation, i.e. `Nothing :-: False`. Conversely, *ar-rafīū* الرَّفِيعُو is in the state `Just True :-: True`. The classical construct state is `Nothing :-: True`. The definite state is `Just _ :-: False`, where _ is `True` for El Dada and Ranta and `False` for Fischer. We may discover that now all the values of `State` are meaningful.[3]

Type declarations are also useful for defining in what categories a given part of speech inflects. For verbs, this is a bit more involved, and we leave it for Figure 2. For nouns, we set this algebraic data type:

```
data ParaNoun = NounS Number Case State
```

In the interpreter, we can now generate all 54 combinations of inflectional parameters for nouns:

```
? [ NounS n c s | n <- enum, c <- enum,
                  s <- values ]
```

The function `values` is analogous to `enum`, and both need to know their type before they can evaluate.

[2] Infix operators can also be written as prefix functions if enclosed in `()`. Functions can be written as operators if enclosed in ``` `` ```. We will exploit this when defining the lexicon's notation.

[3] With `Just False :-: True`, we can annotate e.g. the 'incorrectly' underdetermined *rafīū* رَفِيعُو in *hum-u 'l-masūlūna rafīū 'l-mustawā* هم المسؤولون رَفِيعُو المستوى *they-are the-officials highs-of the-level*, i.e. *they are the high-level officials*.

The 'magic' is that the bound variables `n`, `c`, and `s` have their type determined by the `NounS` constructor, so we need not type anything explicitly. We used the list comprehension syntax to cycle over the lists that `enum` and `values` produce, cf. (Hudak, 2000).

4 ElixirFM Lexicon

Unstructured text is just a list of characters, or string:

```
type String = [Char]
```

Yet words do have structure, particularly in Arabic. We will work with strings as the superficial word forms, but the internal representations will be more abstract (and computationally more efficient, too).

The definition of *lexemes* can include the derivational *root and pattern* information if appropriate, cf. (Habash et al., 2005), and our model will encourage this. The surface word *kitāb* كِتَاب *book* can decompose to the triconsonantal root *k t b* كتب and the morphophonemic pattern `FiCAL` of type `PatternT`:

```
data PatternT = FaCaL | FAL | FaCY |
                FiCAL | FuCCAL | {- ... -}
                MustaFCaL | MustaFaCL
     deriving (Eq, Enum, Show)
```

The `deriving` clause associates `PatternT` with methods for testing equality, enumerating all the values, and turning the names of the values into strings:

```
? show FiCAL → "FiCAL"
```

We choose to build on morphophonemic patterns rather than CV patterns and vocalisms. Words like *istağāb* اِستَجَاب *to respond* and *istağwab* اِستَجوَب *to interrogate* have the same underlying VstVCCVC pattern, so information on CV patterns alone would not be enough to reconstruct the surface forms. Morphophonemic patterns, in this case `IstaFAL` and `IstaFCaL`, can easily be mapped to the hypothetical CV patterns and vocalisms, or linked with each other according to their relationship. Morphophonemic patterns deliver more information in a more compact way. Of course, ElixirFM provides functions for properly interlocking the patterns with the roots:

```
? merge "k t b"  FiCAL    → "kitAb"
? merge "^g w b" IstaFAL  → "ista^gAb"
? merge "^g w b" IstaFCaL → "ista^gwab"
? merge "s ' l"  MaFCUL   → "mas'Ul"
? merge "z h r"  IFtaCaL  → "izdahar"
```

The *izdahar* اِزدَهَر *to flourish* case exemplifies that exceptionless assimilations need not be encoded in the patterns, but can instead be hidden in rules.

The whole generative model adopts the multi-purpose notation of ArabTEX (Lagally, 2004) as a meta-encoding of both the orthography and phonology. Therefore, instantiation of the `"'"` *hamza* carriers or other merely orthographic conventions do not obscure the morphological model. With Encode Arabic[4] interpreting the notation, ElixirFM can at the surface level process the original Arabic script (non-)vocalized to any degree or work with some kind of transliteration or even transcription thereof.

Morphophonemic patterns represent the stems of words. The various kinds of abstract prefixes and suffixes can be expressed either as atomic values, or as literal strings wrapped into extra constructors:

```
data Prefix = Al | LA | Prefix String

data Suffix = Iy | AT | At | An | Ayn |
              Un | In | Suffix String

al = Al; lA = LA   -- function synonyms

aT = AT; ayn = Ayn; aN = Suffix "aN"
```

Affixes and patterns are arranged together via the `Morphs a` data type, where `a` is a triliteral pattern `PatternT` or a quadriliteral `PatternQ` or a non-templatic word stem `Identity` of type `PatternL`:

```
data PatternL = Identity
data PatternQ = KaRDaS | KaRADiS {- ... -}

data Morphs a = Morphs a [Prefix] [Suffix]
```

The word *lā-silkīy* لَاسِلكِي *wireless* can thus be decomposed as the root *s l k* سلك and the value `Morphs FiCL [LA] [Iy]`. Shunning such concrete representations, we define new operators `>|` and `|<` that denote prefixes, resp. suffixes, inside `Morphs a`:

```
? lA >| FiCL |< Iy → Morphs FiCL [LA][Iy]
```

Implementing `>|` and `|<` to be applicable in the intuitive way required Haskell's multi-parameter type classes with functional dependencies (Jones, 2000):

```
class Morphing a b | a -> b where
    morph :: a -> Morphs b

instance Morphing (Morphs a) a where
    morph = id

instance Morphing PatternT PatternT where
    morph x = Morphs x [] []
```

The `instance` declarations ensure how the `morph` method would turn values of type `a` into `Morphs b`.

[4] `http://sf.net/projects/encode-arabic/`

```
|> "k t b" <| [

    FaCaL           'verb'  [ "write", "be destined" ]        'imperf'  FCuL,

    FiCAL           'noun'  [ "book" ]                        'plural'  FuCuL,

    FiCAL |< aT     'noun'  [ "writing" ],

    FiCAL |< aT     'noun'  [ "essay", "piece of writing" ]   'plural'  FiCAL |< At,

    FACiL           'noun'  [ "writer", "author", "clerk" ]   'plural'  FaCaL |< aT
                                                              'plural'  FuCCAL,

    FuCCAL          'noun'  [ "kuttab", "Quran school" ]      'plural'  FaCACIL,

    MaFCaL          'noun'  [ "office", "department" ]        'plural'  MaFACiL,

    MaFCaL |< Iy    'adj'   [ "office" ],

    MaFCaL |< aT    'noun'  [ "library", "bookstore" ]        'plural'  MaFACiL      ]
```

Figure 1: Entries of the ElixirFM lexicon nested under the root *k t b* كتب using morphophonemic templates.

Supposing that `morph` is available for the two types, `(|<)` is a function on `y :: a` and `x :: Suffix` giving a value of type `Morphs b`. The intermediate result of `morph y` is decomposed, and `x` is prepended to the stack `s` of the already present suffixes.

```
(|<) :: Morphing a b =>
        a -> Suffix -> Morphs b

y |< x = Morphs t p (x : s)
    where Morphs t p s = morph y
```

With the introduction of patterns, their synonymous functions and the `>|` and `|<` operators, we have started the development of what can be viewed as a domain-specific language embedded in the general-purpose programming language. Encouraged by the flexibility of many other domain-specific languages in Haskell, esp. those used in functional parsing (Ljunglöf, 2002) or pretty-printing (Wadler, 2003), we may design the lexicon to look like e.g.

```
module Elixir.Data.Lexicon
import Elixir.Lexicon

lexicon = listing  {- lexicon's header -}

 |> {- root one -} <| [ {- Entry a -} ]

 |> {- root two -} <| [ {- Entry b -} ]

 -- other roots or word stems and entries
```

and yet be a verifiable source code defining a data structure that is directly interpretable. The meaning of the combinators `|>` and `<|` could be supplied via an external module `Elixir.Lexicon`, so is very easy to customize. The effect of these combinators might be similar to the `:` and `:-:` constructors that we met previously, but perhaps other data structures might be built from the code instead of lists and pairs.

Individual entries can be defined with functions in a convenient notational form using ``` `` ```. Infix operators can have different precedence and associativity, which further increases the options for designing a lightweight, yet expressive, embedded language.

In Figure 1, each entry reduces to a record of type `Entry PatternT` reflecting internally the lexeme's inherent properties. Consider one such reduction below. Functions like `plural` or `gender` or `humanness` could further modify the `Noun`'s default information:

```
? FiCAL |< aT 'noun' [ "writing" ]  →

  noun (FiCAL |< aT) [ "writing" ]  →

  Entry (Noun [] Nothing Nothing)
        (morph (FiCAL |< aT))
        [ "writing" ]                →

  Entry (Noun [] Nothing Nothing)
        (Morphs FiCAL [] [AT])
        [ "writing" ]
```

The lexicon of ElixirFM is derived from the open-source Buckwalter lexicon (Buckwalter, 2002).[5] We devised an algorithm in Perl using the morpho-

[5] Habash (2004) comments on the lexicon's internal format.

```
data Mood = Indicative | Subjunctive | Jussive | Energetic  deriving (Eq, Enum)
data Gender = Masculine | Feminine                          deriving (Eq, Enum)

data ParaVerb = VerbP       Voice Person Gender Number
              | VerbI Mood Voice Person Gender Number
              | VerbC                    Gender Number       deriving Eq


paraVerbC :: Morphing a b => Gender -> Number -> [Char] -> a -> Morphs b
paraVerbC g n i = case n of

            Singular    -> case g of    Masculine ->  prefix i . suffix ""
                                        Feminine  ->  prefix i . suffix "I"

            Plural      -> case g of    Masculine ->  prefix i . suffix "UW"
                                        Feminine  ->  prefix i . suffix "na"

            _           ->                            prefix i . suffix "A"
```

Figure 2: Excerpt from the implementation of verbal inflectional features and paradigms in ElixirFM.

phonemic patterns of ElixirFM that finds the roots and templates of the lexical items, as they are available only partially in the original, and produces the lexicon in formats for Perl and for Haskell.

Information in the ElixirFM lexicon can get even more refined, by lexicographers or by programmers. Verbs could be declared via indicating their derivational verbal form (that would, still, reduce to some `Morphs a` value), and deverbal nouns and participles could be defined generically for the extended forms. The identification of patterns as to their derivational form is implemented easily with the `isForm` method:

```
data Form = I | II | III | IV {- .. -} XV

? isForm VIII IFtaCaL              → True
? isForm II   TaKaRDuS             → True
? filter (`isForm` MuFCI) [I ..]   → [IV]
```

Nominal parts of speech need to be enhanced with information on the inherent number, gender and humanness, if proper modeling of linguistic agreement in Arabic is desired.[6] Experiments with the Prague Arabic Dependency Treebank (Hajič et al., 2004) show that this information can be learned from annotations of syntactic relations (Smrž, 2007).

5 Morphological Rules

Inferential–realizational morphology is modeled in terms of paradigms, grammatical categories, lexemes and word classes. ElixirFM implements the comprehensive rules that draw the information from the lexicon and generate the word forms given the appropriate morphosyntactic parameters. The whole is invoked through a convenient `inflect` method.

The lexicon and the parameters determine the choice of paradigms. The template selection mechanism differs for nominals (providing plurals) and for verbs (providing all needed stem alternations in the extent of the entry specifications of e.g. Hans Wehr's dictionary), yet it is quite clear-cut (Smrž, 2007).

In Figure 2, the algebraic data type `ParaVerb` restricts the space in which verbs are inflected by defining three Cartesian products of the elementary categories: a verb can have `VerbP` perfect forms inflected in voice, person, gender, number, `VerbI` imperfect forms inflected also in mood, and `VerbC` imperatives inflected in gender and number only.[7]

The paradigm for inflecting imperatives, the one and only such paradigm in ElixirFM, is implemented as `paraVerbC`. It is a function parametrized by some particular value of gender `g` and number `n`. It further takes the initial imperative prefix `i` and the verbal stem (both inferred from the morphophonemic patterns in the lexical entry) to yield the inflected imperative form. Note the polymorphic type of the function, which depends on the following:

```
prefix, suffix :: Morphing a b =>
                  [Char] -> a -> Morphs b

prefix x y = Prefix x >| y
suffix x y = y |< Suffix x
```

[6] Cf. e.g. (El Dada and Ranta, 2006; Kremers, 2003).

[7] Cf. (Forsberg and Ranta, 2004; El Dada and Ranta, 2006).

If one wished to reuse the paradigm and apply it on strings only, it would be sufficient to equate these functions with standard list operations, without any need to reimplement the paradigm itself.

The definition of `paraVerbC` is simple and concise due to the chance to compose with `.` the partially applied `prefix` and `suffix` functions and to virtually omit the next argument. This advanced formulation may seem not as minimal as when specifying the literal endings or prefixes, but we present it here to illustrate the options that there are. An abstract paradigm can be used on more abstract types than just strings.[8] Inflected forms need not be merged with roots yet, and can retain the internal structure:

```
? paraVerbC Feminine Plural "u" FCuL →
  Prefix "u" >| FCuL |< Suffix "na"

? merge "k t b" ({- previous value -}) →

  "uktubna" uktubna اُكْتُبْنَ fem. pl. write!

? [ merge "q r ʼ" (paraVerbC g n "i"
    FCaL) | g <- values, n <- values ] →

  masc.: "iqraʼ" iqraʾ اِقْرَأْ sg. "iqraʼA" iqraʾā
         اِقْرَآ du. "iqraʼUA" iqraʾū اِقْرَؤُوا pl.
  fem.: "iqraʼI" iqraʾī اِقْرَئِي sg. "iqraʼA" iqraʾā
         اِقْرَآ du. "iqraʼna" iqraʾna اِقْرَأْنَ pl. read!
```

The highlight of the Arabic morphology is that the 'irregular' inflection actually rests in strictly observing some additional rules, the nature of which is phonological. Therefore, surprisingly, ElixirFM does not even distinguish between verbal and nominal word formation when enforcing these rules. This reduces the number of paradigms to the prototypical 3 verbal and 5 nominal! Yet, the model is efficient.

Given that the morphophonemic patterns already do reflect the phonological restrictions, the only places of further phonological interaction are the prefix boundaries and the junction of the last letter of the pattern with the very adjoining suffix. The rules are implemented with `->-` and `-<-`, respectively, and are invoked from within the `merge` function:

```
merge :: (Morphing a b, Template b) =>
         [Char] -> a -> [Char]

(->-) :: Prefix -> [Char] -> [Char]
(-<-) :: Char -> Suffix -> [Char]
```

[8] Cf. some morphology-theoretic views in Spencer (2004).

```
'I' -<- x = case x of

  AT      -> "iyaT"  ;   Un      -> "Una"
  Iy      -> "Iy"    ;   In      -> "Ina"

  Suffix ""       -> "i"

  Suffix "Una"    -> "Una"
  Suffix "U"      -> "U"
  Suffix "UW"     -> "UW"

  Suffix "Ina"    -> "Ina"
  Suffix "I"      -> "I"

  Suffix x | x `elem` ["i",  "u"]  -> "I"
           | x `elem` ["iN", "uN"] -> "iN"

        | "n" `isPrefixOf` x ||
          "t" `isPrefixOf` x    -> "I" ++ x

  _       -> "iy" ++ show x
```

`(-<-)` is likewise defined when matching on `'Y'`, `'A'`, `'U'`, and when not matching. `(->-)` implements definite article assimilation and occasional prefix interaction with weak verbs.

Nominal inflection is also driven by the information from the lexicon and by phonology. The reader might be noticing that the morphophonemic patterns and the `Morphs a` templates are actually extremely informative. We can use them as determining the inflectional class and the paradigm function, and thus we can almost avoid other unintuitive or excessive indicators of the kind of weak morphology, diptotic inflection, and the like.

6 Applications and Conclusion

The ElixirFM linguistic model and the data of the lexicon can be integrated into larger applications or used as standalone libraries and resources.

There is another, language-independent part of the system that implements the compilation of the inflected word forms and their associated morphosyntactic categories into morphological analyzers and generators. This part is adapted from (Forsberg and Ranta, 2004). The method used for analysis is deterministic parsing with tries (Ljunglöf, 2002).

ElixirFM also provides functions for exporting and pretty-printing the linguistic model into XML, LaTeX, Perl, SQL, and other custom formats.

We have presented ElixirFM as a high-level functional implementation of Functional Arabic Morphology. Next to some theoretical points, we pro-

posed a model that represents the linguistic data in an abstract and extensible notation that encodes both *orthography* and *phonology*, and whose interpretation is customizable. We developed a domain-specific language in which the lexicon is stored and which allows easy manual editing as well as automatic verification of consistency. We believe that the modeling of both the *written* language and the *spoken* dialects can share the presented methodology.

ElixirFM and its lexicons are open-source software licensed under GNU GPL and available on `http://sf.net/projects/elixir-fm/`.

This work has been supported by the Ministry of Education of the Czech Republic (MSM00216208-38), by the Grant Agency of Charles University in Prague (UK 373/2005), and by the Grant Agency of the Czech Academy of Sciences (1ET101120413).

References

Elsaid Badawi, Mike G. Carter, and Adrian Gully. 2004. *Modern Written Arabic: A Comprehensive Grammar*. Routledge.

Matthew Baerman, Dunstan Brown, and Greville G. Corbett. 2006. *The Syntax-Morphology Interface. A Study of Syncretism*. Cambridge Studies in Linguistics. Cambridge University Press.

Kenneth R. Beesley. 2001. Finite-State Morphological Analysis and Generation of Arabic at Xerox Research: Status and Plans in 2001. In *EACL 2001 Workshop Proceedings on Arabic Language Processing: Status and Prospects*, pages 1–8, Toulouse, France.

Tim Buckwalter. 2002. Buckwalter Arabic Morphological Analyzer Version 1.0. LDC catalog number LDC2002L49, ISBN 1-58563-257-0.

Ali El Dada and Aarne Ranta. 2006. Open Source Arabic Grammars in Grammatical Framework. In *Proceedings of the Arabic Language Processing Conference (JETALA)*, Rabat, Morocco, June 2006. IERA.

Wolfdietrich Fischer. 2001. *A Grammar of Classical Arabic*. Yale Language Series. Yale University Press, third revised edition. Translated by Jonathan Rodgers.

Markus Forsberg and Aarne Ranta. 2004. Functional Morphology. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004*, pages 213–223. ACM Press.

Nizar Habash, Owen Rambow, and George Kiraz. 2005. Morphological Analysis and Generation for Arabic Dialects. In *Proceedings of the ACL Workshop on Computational Approaches to Semitic Languages*, pages 17–24, Ann Arbor, Michigan. Association for Computational Linguistics.

Nizar Habash. 2004. Large Scale Lexeme Based Arabic Morphological Generation. In *JEP-TALN 2004, Session Traitement Automatique de l'Arabe*, Fes, Morocco, April 2004.

Jan Hajič, Otakar Smrž, Petr Zemánek, Jan Šnaidauf, and Emanuel Beška. 2004. Prague Arabic Dependency Treebank: Development in Data and Tools. In *NEMLAR International Conference on Arabic Language Resources and Tools*, pages 110–117. ELDA.

Paul Hudak. 2000. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press.

Mark P. Jones. 2000. Type Classes with Functional Dependencies. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 230–244, London, UK. Springer.

Joost Kremers. 2003. *The Arabic Noun Phrase. A Minimalist Approach*. Ph.D. thesis, University of Nijmegen. LOT Dissertation Series 79.

Klaus Lagally. 2004. ArabTEX: Typesetting Arabic and Hebrew, User Manual Version 4.00. Technical Report 2004/03, Fakultät Informatik, Universität Stuttgart.

Peter Ljunglöf. 2002. *Pure Functional Parsing. An Advanced Tutorial*. Licenciate thesis, Göteborg University & Chalmers University of Technology.

Allan Ramsay and Hanady Mansur. 2001. Arabic morphology: a categorial approach. In *EACL 2001 Workshop Proceedings on Arabic Language Processing: Status and Prospects*, pages 17–22, Toulouse, France.

Otakar Smrž. 2007. *Functional Arabic Morphology. Formal System and Implementation*. Ph.D. thesis, Charles University in Prague.

Abdelhadi Soudi, Violetta Cavalli-Sforza, and Abderrahim Jamari. 2001. A Computational Lexeme-Based Treatment of Arabic Morphology. In *EACL 2001 Workshop Proceedings on Arabic Language Processing: Status and Prospects*, pages 155–162, Toulouse.

Andrew Spencer. 2004. Generalized Paradigm Function Morphology. `http://privatewww.essex.ac.uk/~spena/papers/GPFM.pdf`, October 6.

Gregory T. Stump. 2001. *Inflectional Morphology. A Theory of Paradigm Structure*. Cambridge Studies in Linguistics. Cambridge University Press.

Philip Wadler. 2003. A Prettier Printer. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, Cornerstones of Computing, pages 223–243. Palgrave Macmillan, March 2003.