A Mechanization of Historical Phonology

陳朝陽 진조양

May 2, 2023

Introduction

This project presents an attempt of mechanization of historical phonology. Knowing that not all of my collegues have experiences in computing, this presentation uses high-level pseudo-code to convey the computational content (if the projector happens to work today we may be able to do some living demonstration, we will see).

A high-level overview

The most import constructs we need for such a system:

```
val parse : String → Syllable ;; parse “han” ⇒ an internal
representation of korean syllables
val show : Syllable → String ;; show han (which is an
representaiton of 한) ⇒ “han”
type Rule = ... ;; a representation of phonological rules
val apply : Rule → Syllable → Syllable ;; apply (λ -t . -l)
it ⇒ il
```

Representing segments

```
datatype Place = Labial | Dental | Palatal | Velar | Glottal
datatype Voice = Lax | Aspirated | Tense | Voiced
datatype Manner = Stop | Nasal | Affricate | Fricative
datatype Consonant = Consonant of Voice × Place × Manner

datatype Height = Low | Mid | High
datatype Centrality = Front | Central | Back
datatype Roundedness = Rounded | Unrounded
datatype Vowel = Vowel of Height × Centrality × Rounedness
```

```
datatype τ = Constructor of α × β
;; τ the name of type the ‘datatype’ statement instantiates,
‘Constructor’ is the name of the function that can be used to
construct (duh) such type, and α, β ∈ Type are some types.
;; (one of) the constructor and the constructed type, under
certain conventions may have the same name. Types and
functions (constructors) are not in the same namespace, so
there won’t be collision.
;; ‘of’ is just syntax that indicates the preceding symbol is a
constructor, some languages do not have it, cf. haskell and
its descendants.
datatype Bool = T | F
;; the vertical bar can be read as ‘or’
datatype NP =
  DetP of Det × N
  | NP of N
;; the product operator ‘×’ can be read as ‘and’
```

Encoding Korean Segmental Phonology

```
val p : Consonant = Consonant Lax Labial Stop
;; ‘:’ annotates the type of a term
val t = Consonant Lax Dental Stop
val k = Consonant Lax Velar Stop
...
val ɐ : Vowel = Vowel Low Central Unrounded
val i = Vowel High Front Unrounded
val ɯ = Vowel High Back Unrounded
```

Some Segment-level functions

```
fun tensify (Consonant (_ place manner)) =
    Consonant (Tense place manner)

val seg→seg : segment → segment → segment
fun seg→seg seg-in seg-out =
λ seg . if seg = seg-in then seg-out else seg-in
;; (λ x . x + 1) 1 ⇒ 2
;; (λ x . if x then 1 else 2) T ⇒ 1
```

Example: the sino-korean [-t] → [-l]

NB. this is not really a sound change in (sino-)korean, but it serves as a decent example.

```
fun t→l (onset, nucleus, coda) =
let val change = seg→seg /t/ /l/
in (onset, nucleus, change coda)
;; t→l it ⇒ il ;; 일
;; t→l chit ⇒ chil ;; 칠
```

Coda neutralization

```
fun coda-neutralize (Consonant (_ place Stop)) =
Consonant Lax place Stop
```

Representing syllables

```
datatype Onset = Onset of List Consonant
datatype Nucleus = Nucleus of List Vowel
datatype Coda = Coda of List Consonant
datatype Syllable = Syllable of Onset × Nucleus × Coda
;; this might be unnecessarily verbose
datatype Syllable =
Syllable of List Consonant × List Vowel × List Consonant
;; one can simply do this, the difference is negligible
```

Silencing of initial [ŋ-] → Ø

One of the motivation to define syllabic structure is to express deletion. Note that Ø is not really a consonant (nor is it any syllable).

Representing sound changes

```
type Rule = Syllable → Syllable
;; this alone might be enough
type Pred = Syllable → Bool
type Change = Syllable → Syllable
datatype Rule = Rule of Pred × Change
```

Some Extralinguistic Constructs

```
datatype List α = Nil | Cons of α × List
alias Nil = []
infix x :: xs = Cons x xs
;; [1,2,3] ::= 1 :: 2 :: 3 :: Nil

fun map f [] = []
  | map f (x::xs) = (f x) :: map f xs

fun foldl f x [] = x
  | foldl f x (y::ys) = foldl f (f y x) ys
```

```
type Syllable = ...
type Pword = List Syllable
;; unsuprisingly, a phonological word is a list of syllables

map (λ x . x + 1) [1,2,3] ⇒ [2,3,4]
map rule [s1, s2, s3] ⇒ [s1', s2', s3']
;; map is used to apply a (syllable-level) rule to a list of
syllable (pword)

foldl apply etymon [sc1, sc2, sc3] ⇒ reflex
;; foldl is used to apply a list of (pword-level) sound changes
to an etymon, subsequently deriving its reflex
```

Di-syllabic rule: assimilation

```
fun disyllabic-change pred change pword =
case pword of
| [] ⇒ []
| [x] ⇒ [x]
| x :: y :: tl ⇒
if pred (x,y)
then let val (x',y') = change (x,y)
     in x' :: disyllabic-change pred change (y' :: tl)
else x :: disyllabic-change pred change (y :: tl)
```

where pred : Syllable $\times$ Syllable $\rightarrow$ Bool, and change : Syllable $\times$ Syllable $\rightarrow$ Syllable $\times$ Syllable.

```
val nasal-assimilation =
let val pred = “...”
    val change = “...”
;; they are better explained in a natural language
in disyllabic-change pred change

nasal-assimilation [sip, njʌn] ⇒ [sim, njʌn]
;; 십년 十年
```

Resyllabification

Resyllabification can be implemented using ‘disyllabic-change’. This is either left as an exercise for the curious reader (or will be demonstrated by me if we have enough time).

Putting it all together: the word problem

So far we have talked about how to define segments and syllables, and how to define and apply sound changes. This is where I stopped last time (for Latin-Spanish). But historical phonology is more than this, considerring the following problem:

The word problem

Given two strings $S, S' \in \Sigma^*$, where Σ ($\varnothing \in \Sigma$) is an alphabet and Σ^* strings closed under concatenation, and a bounded collection of rewrite rules Λ ($\mathtt{id} \in \Lambda$), decide whether $S \to_\Lambda^n S'$ in n steps for some $n \in \mathbb{N}$ and show (all the) routes that reduces S to S'.

ibid., in normal people's speech

Given a Middle Korean (phonological) word, K, and a Modern Korean word, $K' \in \Sigma^*$ where Σ is the phonological inventory of Korean (throughout history) and Σ^* is the set that contains all possible (not necessarily attested) Koreanic words (throughout history), and a collection of phonological rules (sound changes), decide whether K can be reduced to K' and show all the possible ways of reducing K to K'.

Σ = {p, $\mathrm{p^h}$, b, m ... ɒ, ɯ, ɛ ... l, ɾ}
Σ^* = {syera-kuy, psal, ... palam}
Λ = {ŋ $\rightarrow \varnothing$, jʌ $\rightarrow$ ʌ, ... }
where $[\![\lambda]\!]$ = map λ

This is a rough sketch.

Solvability of the word problem

The word problem has been proved unsolvable (undeterministic) for many cases. In our particular case (historical phonology), I can give the following sketch about how to construct a decidable system:

- rewriting rules can not have cyclicity
- rule sets will have to be decreasing throughout the reduction process (these two things are roughly equivalent in our case)

cyclicity

Λ = l$\rightarrow$n, n$\rightarrow$l
S, S' = nala

Rough Estimate of Time-complexity

Since without the constraints mentioned above, such system is not decidable, let's impose an ad-hoc constraint that would make it decidable: each rule can only applied once. This makes the worse case factorial, which is pretty bad, even considering that the amount of reconstructed phonological rules for a well-documented language is usually under 100 (check out what is $20!$). The 'average' case that I can construct is around $\frac{1}{2^{\Sigma log2(n)}} n^{log2(n)+1}$ (with each iteration roughly discarding half of the rules), which is of course still super-polynomial, but much slower than factorial.