The isiZulu verbalisation algorithms: design and documentation

C. Maria Keet

Department of Computer Science, University of Cape Town, South Africa
mkeet@cs.uct.ac.za

May 27, 2018

Abstract

Automatically generating text in isiZulu—the largest language by first language speakers in South Africa—has been investigated over the past few years. This was done in an incremental fashion, covering one feature at a time. The principal three components are generating plurals from a noun in the singular, the main axiom types in an ontology, such as subsumption of named classes and existential quantification, and the phonological conditioning that is required in certain cases. This document lists the core algorithms and contains brief explanatory descriptions, which serve as implementation-independent documentation of the code and to describe those algorithms not included in the respective papers.
The code is available as a downloadable zip file from the project website at `http://www.meteck.org/geni/`, with as cut-off date the state in February 2018.

Keywords: Natural Language Generation, Controlled Natural Language, Pluralisation, Phonological Conditioning, isiZulu, OWL

Contents

1 Pluralisation of nouns

The algorithm to pluralise nouns in the singular included here (Algorithm 1) is slightly changed cf. the one described in [1] and implemented in its related supplementary material, for there it was a stand-alone file and read in the test sets, whereas here that 'wrapping' is omitted.

Algorithm 1 Pluralise isiZulu noun (full version)

```
procedure PLURALISE(n)
input: noun in singular                              {note: n noun, nc noun class, p plural}
if n in exceptionList then
    p ← plural exception                             {check exceptions list}
else
    nc ← getNC(n)                                    {lookup noun class of noun n}
                                                     {pluralise compound nouns}
    if ' ' ∈ n and not endswith(nc, m) then
        main, rest ← split(n)      {to pluralise main noun and modify the modifier word (rest)}
        switch
            case nc == 1 and rest[0] ∉ {a, e, i, o, u}
                rest′ = b + rest[1:]
            case nc == 1 and rest[0] ∈ {a, e, i, o, u}
                rest′ = aba + rest[1:]
            case nc == 3a
                rest′ = aba + rest[1:]
            case (nc == 9 or nc == 7) and rest[0] ∉ {a, e, i, o, u}
                rest′ = z + rest[1:]
            case (nc == 9 or nc == 7) and rest[0] ∈ {a, e, i, o, u}
                rest′ = rest[0] + zi + rest[3:]
                break                                {cases for other ncs are yet to be investigated}
        end switch
        p′ ← pluralise(main)
        p ← p′ + ' ' + rest′
    else
                                                     {pluralise the regular nouns}
        switch
            case (startswith(n, um) or startswith(n, uM)) and n[2]!= u and nc == 1
                p = aba + n[2:]
            case (startswith(n, um) or startswith(n, uM)) and n[2] ∈ {a, e, i, o}
                p = ab + n[2:]
            case startswith(n, umu) and nc == 1
                p = aba + n[3:]
            case startswith(n, u) and nc == 1a or nc == 3a
                p = o + n[1:]
            case startswith(n, um) and n[2] != u and nc == 3
                p = imi + n[2:]
            case startswith(n, umu) and nc == 3
                p = imi + n[3:]
```

```
        case startswith(n, i) and n[0:2] != ili and (nc == 5 or nc == 9a)
            p = ama + n[1:]
        case startswith(n, ili) and nc == 5
            p = ama + n[3:]
        case startswith(n, isi) and nc == 7
            p = izi + n[3:]
        case startswith(n, is) and nc == 7 and n[2] ∈ {a, e, o, u}
            p = iz + n[2:]
        case startswith(n, im) and nc == 9
            p = izi + n[1:]
        case (startswith(n, in) or startswith(n, iN)) and nc == 9
            p = izin + n[2:]
        case startswith(n, ulu) and nc == 11
            p = izi + n[3:]
        case startswith(n, u) n[1:2] != lu and nc == 11
            p = izi + n[1:]
        case (startswith(n, ubu) and nc == 14
            p = n
        case (startswith(n, uku) or startswith(n, uk)) and (nc == 15 or nc == 17)
            p = n                                        {nc15 and nc17 don't pluralise}
        case endswith(nc, m)
            p ← n                                        {mass nouns don't pluralise}
        case nc ∈ {2, 4, 6, 8, 2a, 10}                  {noun exists only in plural form}
            p = n
            break
      end loop
    end if
end if
return p
end procedure
```

2 Verbalisation algorithms for (almost) $\mathcal{ALC}$

The following list of algorithms are included in this document, which were first described in [2, 3] and more comprehensively in [6]:

- Simple taxonomic subsumption, i.e., named class subsumption of the axiom type $C \sqsubseteq D$, in Algorithm 2;
- Simple existential quantification with named classes, i.e., of the axiom type $C \sqsubseteq \exists R.D$, in Algorithm 3;
- Negation in an axiom, covering both the axiom types $C \sqsubseteq \neg D$ and $C \sqsubseteq \neg \exists R.D$, in Algorithm 4. This algorithm has been updated with vowel-commencing verb roots cf. the one presented in [6].

Algorithm 3 and Algorithm 4 have functions to look up things from a list. They are the 'lookup tables' for noun classes, and their corresponding quantitative, relative, and (negative) subject concords, which are included in Table 1 for easy readable reference.

Table 1: Zulu noun classes with examples and a selection of concords. NC: Noun class; PRE: prefix; QC: quantitative concord; RC: relative concord; SC: subject concord; NEG SC: negative subject concord; PC: possessive concord. Updated cf. the tables in [1, 5, 6] (deviant cases of prefixes not included).

NC	Full PRE	QC (∀)	RC	QC (∃)	SC	NEG SC	PC
1	um(u)-	wonke	o-	ye-	u-	aka-	wa-
2	aba-	bonke	aba-	bo-	ba-	aba-	ba-
1a	u-	wonke	o-	ye-	u-	aka-	wa-
2a	o-	bonke	aba-	bo-	ba-	aba-	ba-
3a	u-	wonke	o-	ye-	u-	aka-	wa-
2a	o-	bonke	aba-	bo-	ba-	aba-	ba-
3	um(u)-	wonke	o-	wo-	u-	awu-	wa-
4	imi-	yonke	e-	yo-	i-	ayi-	ya-
5	i(li)-	lonke	eli-	lo-	li-	ali-	la-
6	ama-	onke	a-	wo-	a-	awa-	a-
7	isi-	sonke	esi-	so-	si-	asi-	sa-
8	izi-	zonke	ezi	zo-	zi-	azi-	za-
9a	i-	yonke	e-	yo-	i-	ayi-	ya-
6	ama-	onke	a-	wo-	a-	awa-	a-
9	i(n)-, i(m)-	yonke	e-	yo-	i-	ayi-	ya-
10	izi(n)-, izi(m)-	zonke	ezi-	zo-	zi-	azi-	za-
11	u(lu)-	lonke	olu-	lo-	lu-	alu-	lwa-
10	izi(n)-, izi(m)-	zonke	ezi-	zo-	zi-	azi-	za-
14	ubu-	bonke	obu-	bo-	bu-	abu-	ba-
15	uku-	konke	oku-	ko-	ku-	aku-	kwa-
17	ku-	lonke	olu-	lo-	lu-		kwa-

Further, note that these algorithms require pluralisation of the head noun, whose algorithm is included in Section 1 (Algorithm 1).

There are specific cases with the part-whole relations, which are described in Section 3. Both they and 'regular irregular' verbs require phonological conditioning, which is described in Section 4.

Algorithm 2 (TaxSubs) Verbalisation of taxonomic subsumption, named classes ($C \sqsubseteq D$).

Require: $\mathcal{C}$ set of classes, language $\mathcal{L}$ with $\sqsubseteq$ for subsumption and $\neg$ for negation; variables: A axiom, NC_i nounclass, $c_1, c_2 \in \mathcal{C}$, a_1 term, a_2 letter; functions: $getFirstClass(A)$, $getSecondClass(A)$, $getNC(C)$, $checkNegation(A)$, $getFirstChar(C)$.

Require: axiom A with a $\sqsubseteq$ has been retrieved **and** named classes on the lhs and rhs

$c_1 \leftarrow getFirstClass(A)$ {get subclass}
$c_2 \leftarrow getSecondClass(A)$ {get superclass}
$NC_1 \leftarrow getNC(c_1)$ {determine noun class by augment and prefix or dictionary}
$NC_2 \leftarrow getNC(c_2)$ {determine noun class by augment and prefix or dictionary}
if $checkNegation(A) == true$ **then**
    {use $negation$ (Algorithm 4)}
else
    $a_2 \leftarrow getFirstChar(c_2)$ {retrieve first letter of c_2}
    switch
        case $a_2 =$ 'i' **then**
            RESULT $\leftarrow$ 'c_1 yc_2.' {verbalise as taxonomic subsumption with y}
        case $a_2 =$ {'a', 'o', 'u'} **then**
            RESULT $\leftarrow$ 'c_1 ngc_2.' {verbalise as taxonomic subsumption with ng}
        case $a_2 \notin$ {'a', 'i', 'o', 'u',} **then**
            RESULT $\leftarrow$ '*this is not a well-formed isiZulu noun.*'
    end switch
end if
return RESULT

Algorithm 3 (AllSome) Verbalisation of "all-some" axiom type ($C \sqsubseteq \exists R.D$)

Require: $\mathcal{C}$ set of classes, language $\mathcal{L}$ with $\sqsubseteq$ for subsumption and $\exists$ for existential quantification; variables: A axiom, NC_i noun class, $c_1, c_2 \in \mathcal{C}$, $o \in \mathcal{R}$, a_1 a term; r_2, q_2 concords; functions: $getFirstClass(A)$, $getSecondClass(A)$, $getNC(C)$, $getRC(NC_i)$, $getQC(NC_i)$, $getVSofOP(o)$.

Require: axiom A with a $\sqsubseteq$ has been retrieved **and** an $\exists$ on the rhs of the inclusion

$c_1 \leftarrow getFirstClass(A)$ {get subclass}
$c_2 \leftarrow getSecondClass(A)$ {get superclass}
$o \leftarrow getObjProp(A)$ {get object property}
$v \leftarrow getVSofOP(o)$ {get verb stem of object property}
$NC_1 \leftarrow getNC(c_1)$ {determine noun class by augment and prefix or dictionary}
$NC_2 \leftarrow getNC(c_2)$ {determine noun class by augment and prefix or dictionary}
$NC'_1 \leftarrow$ lookup plural nounclass of NC_1 {from known list}
$c'_1 \leftarrow pluralise(c_1, NC'_1)$ {call algorithm $pluralise$ to generate a plural from o}
$a_1 \leftarrow$ lookup quantitative concord for NC'_1 {from quantitative concord (QC(all)) list}
$r_2 \leftarrow getRC(NC_2)$ {get relative concord for c_2 from the QC$_{\text{dwa}}$-list}
$q_2 \leftarrow getQC(NC_2)$ {get quantitative concord for c_2 from the QC$_{\text{dwa}}$-list}
if $checkNegation(A) == true$ **then**
    {use $negation$ (Algorithm 4)}
else
    if o annotated with present tense **then**
        $conj_{nc1} \leftarrow$ lookup SC of NC'_1 {from known SC list}
        $o' \leftarrow conj_{nc1} v$ {generate conjugated verb}
        RESULT $\leftarrow$ 'a_1 c'_1 o'a c_2 $r_2 q_2$dwa.' {verbalise the axiom}
    else
        RESULT $\leftarrow$ '*passive voice and inverses are not supported yet.*'
    end if
end if
return RESULT

Algorithm 4 (Negation) Verbalisation of negation in an axiom, as disjointness or negated object property (i.e., axioms of type $C \sqsubseteq \neg D$ and $C \sqsubseteq \neg\exists R.D$).

Require: $\mathcal{C}$ set of classes, language $\mathcal{L}$ with $\sqsubseteq$ for subsumption and $\neg$ for negation; variables: A axiom, NC_i noun class, $c_1, c_2 \in \mathcal{C}$, a_1 term, a_2 letter and n, p are concords, v verb stem; functions: $checkNegation(A)$, $getNSC(NC_i)$, $getPNC(NC_i)$.

Require: $checkNegation(A) == true$

if negation directly preceded by $\sqsubseteq$ **and** directly followed by c_2 **then**
  $NC_1' \leftarrow$ lookup plural nounclass of NC_1 {from known list}
  $c_1' \leftarrow pluralise(c_1, NC_1')$ {call algorithm $pluralise$ to generate a plural from o}
  $a_1 \leftarrow$ lookup quantitative concord for NC_1' {from quantitative concord (QC(all)) list}
  $n \leftarrow getNSC(NC_1')$ {get negative subject concord for c_1'}
  $p \leftarrow getPNC(NC_2)$ {get pronomial for c_2}
  RESULT $\leftarrow$ 'a_1 c_1' np c_2.' {verbalise the disjointness (a_1 is QC(all))}
else if negation in front of OP **then**
  $v' \leftarrow$ remove final vowel of v {i.e., obtain the (possibly extended) verb root}
  $n \leftarrow getNSC(NC_1')$ {get negative subject concord for c_1'}
  if $v' \in \{a, e, i, o, u,\}$ **then**
    $negv \leftarrow phonoCondNegSc(v', n)$
  else
    $negv \leftarrow n + v'$
  end if
  RESULT $\leftarrow$ 'a_1 c_1' $negv$i c_2 $r_2 q_2$dwa.' {verbalise the axiom}
else {negation in front of c_2 **and** A contains an OP}
  RESULT $\leftarrow$ '*verbalisation of this class negation is not supported yet.*'
end if
return RESULT

3 Algorithms for basic part-whole relations

The algorithms are here presented as functions that integrate with the other algorithms presented in the preceding sections, in the sense that only the "all some" the axiom type is considered, i.e., $C \sqsubseteq \exists R.D$, where in these cases, R is the 'has part' or the 'part of' reading direction. For rationale and descriptions of the verbalisation patterns, see the corresponding paper [5]; an informal summary of the part-whole relations is shown in Figure 1. The 'has part' direction algorithm (Algorithm 5) was first published in [5].

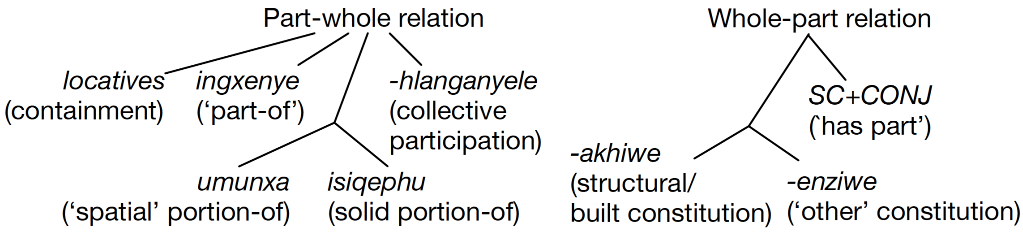

Figure 1: Preliminary taxonomy based on the verbalisation patterns in [5] (source: [4]).

3.1 Whole-part reading direction

Algorithm 5 Determine the verbalisation of basic whole-part in an axiom. This covers the structural, involvement, containment, membership, part-subquantities, and participation whole-part relations

Require: $\mathcal{C}$ set of classes, language $\mathcal{L}$, $\sqsubseteq$ for subsumption, $\exists$ for existential quantification; variables: A axiom, NC_i noun class, $w, p \in \mathcal{C}$, $o \in \mathcal{R}$, a_w a term; r_p, q_p concords;
Require: axiom of the form $W \sqsubseteq \exists wp.P$ has been retrieved for verbalisation

$w \leftarrow getFirstClass(A)$ {get whole}
$p \leftarrow getSecondClass(A)$ {get part}
$wp \leftarrow getObjProp(A)$ {get wp type ('default' parthood here)}
$NC_w \leftarrow getNC(w)$ {obtain noun class whole}
$NC_p \leftarrow getNC(p)$ {obtain noun class part}
$w_{pl} \leftarrow pluralise(w, NC_w)$ {generate plural, using the pluraliser algorithm}
$NC'_w \leftarrow getPlNC(NC_w)$ {obtain plural NC, from known list}
$a_w \leftarrow getQCAll(NC'_w)$ {obtain quantitative concord (QC(all))}
$s_w \leftarrow getSC(NC'_w)$ {obtain subject concord}
$conjp \leftarrow phonoCondition($'na'$,p)$ {prefix P with the CONJ, phonologically conditioned}
$r_p \leftarrow getRC(NC_p)$ {obtain relative conc. for p}
$q_p \leftarrow getQC(NC_p)$ {obtain quant. concord for p from the QC (exists)-list}
RESULT $\leftarrow$ ' a_w w_{pl} $s_w conjp$ $r_p q_p$dwa. ' {verbalise the simple axiom}
return RESULT

Because there is quite some duplication, like fetching the classes, pluralising, and adding the quantitative concords, we put this now in a separate algorithm, *commonFunctWP*, being Algorithm 6, that will be called by all the other functions. In some cases, it fetches a bit more than strictly needed (e.g., an RC and QC too much), but it saves a lot of duplication in the presentation here, and it's not computationally costly (linear, with a small list). The solid portions deviate from this, due to mostly dealing with a noun phrase (e.g., 'sample of blood'), so it is written in full there (Algorithm 10).

Algorithm 6 Common functions for wp verbalisation, *commonFunctWP*.

Require: $\mathcal{C}$ set of classes, language $\mathcal{L}$, $\sqsubseteq$ for subsumption, $\exists$ for existential quantification; variables: A axiom, NC_i noun class, $w, p \in \mathcal{C}$, $o \in \mathcal{R}$, a_w a term; r_p, q_p concords;
Require: axiom of the form $W \sqsubseteq \exists wp.P$ has been retrieved for verbalisation

$w \leftarrow getFirstClass(A)$ {get whole}
$p \leftarrow getSecondClass(A)$ {get part}
$wp \leftarrow getObjProp(A)$ {get wp type}
$NC_w \leftarrow getNC(w)$ {obtain noun class whole}
$NC_p \leftarrow getNC(p)$ {obtain noun class part}
$w_{pl} \leftarrow pluralise(w, NC_w)$ {generate plural, using the pluraliser algorithm}
$NC'_w \leftarrow getPlNC(NC_w)$ {obtain plural NC, from known list}
$a_w \leftarrow getQCAll(NC'_w)$ {obtain quantitative concord (QC(all))}
$s_w \leftarrow getSC(NC'_w)$ {obtain subject concord}
$conjp \leftarrow phonoCondition($'na'$,p)$ {prefix P with the CONJ, phonologically conditioned}
$r_p \leftarrow getRC(NC_p)$ {obtain relative conc. for p}
$q_p \leftarrow getQC(NC_p)$ {obtain quant. concord for p from the QC (exists)-list}

Algorithm 7 Determine the verbalisation of basic whole-part in an axiom. Specifically: wp for spatial portions, without *-dwa*. (`wp_spatial`)

input: two named classes that have the role w and p, respectively
commonFunctWP(w,p)
if $wp ==$ spatial portion **then**
  RESULT $\leftarrow$ ‘ a_w w_{pl} $s_w conjp$.’ {verbalise the axiom}
end if
return RESULT

Algorithm 8 Determine the verbalisation of basic whole-part in an axiom. Specifically: participation with collectives, and w in singular (`wp_cp`)

input: two named classes that have the role w and p, respectively
commonFunctWP(w,p)
if $wp ==$ collective participation **then**
  $a_w \leftarrow getQCAll(NC_w)$ {obtain quantitative concord (QC(all))}
  $s_w \leftarrow getSC(NC_w)$ {obtain subject concord}
  RESULT $\leftarrow$ ‘ a_w w $s_w conjp$ $r_p q_p$dwa.’ {verbalise the axiom}
end if
return RESULT

Algorithm 9 Determine the verbalisation of basic whole-part in an axiom. That is: subquantities [as parts] in singular, and no *-dwa* (`wp_s`)

input: two named classes that have the role w and p, respectively
commonFunctWP(w,p)
if $wp ==$ subquantities **then**
  $a_w \leftarrow getQCAll(NC_w)$ {obtain quantitative concord (QC(all))}
  $s_w \leftarrow getSC(NC_w)$ {obtain subject concord}
  RESULT $\leftarrow$ ‘ a_w w $s_w conjp$.’ {verbalise the simple axiom}
end if
return RESULT

Algorithm 10 Determine the verbalisation of basic whole-part in an axiom. Specifically: solid portion has W in singular, and the P with the PC, assuming that the part-quantity component is one word only (`wp_solid_p`).

Require: axiom of the form $W \sqsubseteq \exists wp.P$ has been retrieved for verbalisation

$w \leftarrow getFirstClass(A)$ {get whole}
$p \leftarrow getSecondClass(A)$ {get part}
$wp \leftarrow getObjProp(A)$ {get wp type}
if $wp ==$ solid portion **then**
  $NC_w \leftarrow getNC(w)$ {obtain noun class whole}
  $q \leftarrow$ first word of p {p is typically a noun phrase or compound noun, first part the quantity, like slice, sample, etc}
  if length(p) $== 2$ **then**
    $stuff \leftarrow$ second word of p
  else
    $stuff \leftarrow$ remainder of p
  end if
  $NC_q \leftarrow getNC(q)$ {obtain noun class quantity}
  $a_w \leftarrow getQCAll(NC_w)$ {obtain quantitative concord (QC(all))}
  $s_w \leftarrow getSC(NC_w)$ {obtain subject concord}
  $conjp \leftarrow phonoCondition($'na'$,q)$ {prefix the quantity-part of P with the CONJ, phonologically conditioned}
  $pc_q \leftarrow getPC(q)$ {obtain possessive conc. for q, for the 'of'}
  $os \leftarrow phonoCondition(pc_q, stuff)$ {generate "of stuff"}
  $r_p \leftarrow getRC(NC_q)$ {obtain relative conc. for p}
  $q_p \leftarrow getQC(NC_q)$ {obtain quant. concord for p from the QC (exists)-list}
  RESULT $\leftarrow$ ' a_w w $s_w conjp$ os $r_p q_p$dwa.' {verbalise the axiom}
end if
return RESULT

Algorithm 11 Determine the verbalisation of basic whole-part in an axiom. Specifically: constitution, of the built type (renamed this function after the inlg16). (`const_a`)

Require: axiom of the form $W \sqsubseteq \exists wp.P$ has been retrieved for verbalisation

$w \leftarrow getFirstClass(A)$ {get whole}
$p \leftarrow getSecondClass(A)$ {get part}
$wp \leftarrow getObjProp(A)$ {get wp type}
if $wp ==$ built constitution **then**
  $NC_w \leftarrow getNC(w)$ {obtain noun class whole}
  $w_{pl} \leftarrow pluralise(w, NC_w)$ {generate plural, using the pluraliser algorithm}
  $NC'_w \leftarrow getPlNC(NC_w)$ {obtain plural NC, from known list}
  $a_w \leftarrow getQCAll(NC'_w)$ {obtain quantitative concord (QC(all))}
  $sv \leftarrow phonoCondVerb($'akhiwe'$, NC'_w)$ {add SC + phono. cond. for vowel-commencing verbs}
  $op \leftarrow phonoCondition($'nga'$,p)$ {generate "of part"}
  RESULT $\leftarrow$ ' a_w w_{pl} sv op. ' {verbalise the axiom}
end if
return RESULT

Algorithm 12 Determine the verbalisation of basic whole-part in an axiom. Specifically: constitution as well, for other 'non-construction' constitution. (`const_e`)

Require: axiom of the form $W \sqsubseteq \exists wp.P$ has been retrieved for verbalisation

$w \leftarrow getFirstClass(A)$ {get whole}
$p \leftarrow getSecondClass(A)$ {get part}
$wp \leftarrow getObjProp(A)$ {get wp type}
if $wp ==$ the other constitution **then**
  $NC_w \leftarrow getNC(w)$ {obtain noun class whole}
  $w_{pl} \leftarrow pluralise(w, NC_w)$ {generate plural, using the pluraliser algorithm}
  $NC'_w \leftarrow getPlNC(NC_w)$ {obtain plural NC, from known list}
  $a_w \leftarrow getQCAll(NC'_w)$ {obtain quantitative concord (QC(all))}
  $sv \leftarrow phonoCondVerb($'enziwe'$, NC'_w)$ {add SC + phono. cond. for vowel-commencing verbs}
  $op \leftarrow phonoCondition($'nga'$, p)$ {generate "of part"}
  RESULT $\leftarrow$ ' $a_w\ w_{pl}\ sv\ op.$ ' {verbalise the axiom}
end if
return RESULT

3.2 Part-whole reading direction

For the sake of presentation, also here we put the common functions in a separate algorithm tha tis used by the others (Algorithm 13).

Algorithm 13 Common functions for pw verbalisation, *commonFunctPW*.

Require: $\mathcal{C}$ set of classes, language $\mathcal{L}$, $\sqsubseteq$ for subsumption, $\exists$ for existential quantification; variables: A axiom, NC_i noun class, $w, p \in \mathcal{C}$, $o \in \mathcal{R}$, a_w a term; r_p, q_p concords;
Require: axiom of the form $P \sqsubseteq \exists pw.W$ has been retrieved for verbalisation

$p \leftarrow getFirstClass(A)$ {get whole}
$w \leftarrow getSecondClass(A)$ {get part}
$pw \leftarrow getObjProp(A)$ {get pw type}
$NC_p \leftarrow getNC(p)$ {obtain noun class whole}
$NC_w \leftarrow getNC(w)$ {obtain noun class part}
$p_{pl} \leftarrow \textit{pluralise}(p, NC_p)$ {generate plural, using the pluraliser algorithm}
$NC'_p \leftarrow getPlNC(NC_p)$ {obtain plural NC, from known list}
$a_p \leftarrow getQCAll(NC'_p)$ {obtain quantitative concord (QC(all))}
$s_p \leftarrow getSC(NC'_p)$ {obtain subject concord}
$r_w \leftarrow getRC(NC_w)$ {obtain relative conc. for w}
$q_w \leftarrow getQC(NC_w)$ {obtain quant. concord for w from the QC (exists)-list}

Algorithm 14 Determine the verbalisation of basic part-whole in an axiom. Specifically: structural, involvement, membership, part-subquantities, participation, part-whole relations. (`pw`)

input: two named classes that have the role w and p, respectively
commonFunctPW(p,w)
if $pw ==$ generic part **then**
  $pc =$ 'ya' {no look-up needed for the PC, because it's always ya- because always ingxenye (nc9)}
  $pcw \leftarrow phonoCondition(pc, w)$
  RESULT $\leftarrow$ ' a_p p_{pl} $s_p$yingxenye pcw $r_w q_w$dwa. ' {verbalise the axiom}
end if
return RESULT

Algorithm 15 Determine the verbalisation of basic part-whole in an axiom. Specifically: part-whole, in the singular as well, to cater for subquantities that can be both mass and count noun, depending on context. (`pw_s`)

input: two named classes that have the role w and p, respectively
commonFunctPW(p,w)
if pw == subquantity of **then**
  $a_p \leftarrow getQCAll(NC_p)$ {obtain quantitative concord (QC(all))}
  $s_p \leftarrow getSC(NC_p)$ {obtain subject concord}
  pc = 'ya' {no look-up needed for the PC, because it's always ya- because always ingxenye (nc9)}
  $pcw \leftarrow phonoCondition(pc,w)$
  RESULT ← ' a_p p $s_p$yingxenye pcw. ' {verbalise the axiom}
end if
return RESULT

Algorithm 16 Determine the verbalisation of basic part-whole in an axiom. Specifically: solid portion-of. (`pw_solid_p`)

Require: axiom of the form $P \sqsubseteq \exists pw.W$ has been retrieved for verbalisation
$p \leftarrow getFirstClass(A)$ {get whole}
$w \leftarrow getSecondClass(A)$ {get part}
$pw \leftarrow getObjProp(A)$ {get pw type}
if pw == solid portion **then**
  $q \leftarrow$ first word of p {p is typically a noun phrase or compound noun, first part the quantity, like slice, sample, etc}
  if length(p) == 2 **then**
    $stuff \leftarrow$ second word of p
  else
    $stuff \leftarrow$ remainder of p
  end if
  $NC_q \leftarrow getNC(q)$ {obtain noun class quantity}
  $NC_w \leftarrow getNC(w)$ {obtain noun class whole}
  $NC'_q \leftarrow getPlNC(NC_q)$
  $a_q \leftarrow getQCAll(NC'_q)$ {obtain quantitative concord (QC(all))}
  $pc_q \leftarrow getPC(q)$
  $os \leftarrow phonoCondition(pc_q, stuff)$
  $q_{pl} \leftarrow pluralise(q, NC'_q)$
  $s_q \leftarrow getSC(NC'_q)$
  pc = 'sa' {no look-up needed for the PC, because it's always sa- because always isiqephu (nc7)}
  $pcw \leftarrow phonoCondition(pc,w)$
  $r_w \leftarrow getRC(NC_w)$ {obtain relative conc. for w}
  $q_w \leftarrow getQC(NC_w)$ {obtain quant. concord for w from the QC (exists)-list}
  RESULT ← ' a_q q_{pl} os $s_q$yisiqephu pcw $r_w q_w$dwa.' {verbalise the axiom}
end if
return RESULT

Algorithm 17 Determine the verbalisation of basic part-whole in an axiom. Specifically: spatial portion-of. (`pw_spatial_p`)

input: two named classes that have the role w and p, respectively
commonFunctPW(p,w)
if $pw ==$ spatial portion of **then**
  $pc =$ 'wa' {no look-up needed for the PC, because it's always wa- because always umunxa (nc3)}
  $pcw \leftarrow phonoCondition(pc,w)$
  RESULT ← ' a_p p_{pl} $s_p$ngumunxa pcw. ' {verbalise the axiom}
end if
return RESULT

Algorithm 18 Determine the verbalisation of basic part-whole in an axiom. Specifically: participates-in, for collective parts, in singular. (pw_pi_c)

input: two named classes that have the role w and p, respectively
commonFunctPW(p,w)
if $pw ==$ collective participates in **then**
  $a_p \leftarrow getQCAll(NC_p)$ {obtain quantitative concord (QC(all))}
  $s_p \leftarrow getSC(NC_p)$ {obtain subject concord}
  $lpre \leftarrow phonoCondLocPrefix(w,\ NC_w)$
  {add locative prefix to whole (if nc = 1a, 2a, 3a, or 17 then ku+word, else e+word)}
  $lpreWlsuf \leftarrow phonoCondLocSuffix(lpre)$ {add locative suffix to whole (the -ini/-eni/-wini etc)}
  RESULT ← ' a_p p $s_p$hlanganyele $lpreWlsuf$ $r_w q_w$dwa.' {verbalise the axiom}
end if
return RESULT

Algorithm 19 Determine the verbalisation of basic part-whole in an axiom. Specifically: contained-in. (`pw_ci`)

input: two named classes that have the role w and p, respectively
commonFunctPW(p,w)
if $pw ==$ contained in **then**
  $Wlsuf \leftarrow phonoCondLocSuffix(w)$ {add locative suffix to whole (the -ini/-eni/-wini etc)}
  $lpreWlsuf \leftarrow phonoCondLocPrefix(Wlsuf,\ NC_w)$
  {add locative prefix to whole (if nc = 1a, 2a, 3a, or 17 then ku+word, else e+word)}
  RESULT ← ' a_p p_{pl} s_ps$lpreWlsuf$ $r_w q_w$dwa.' {verbalise the axiom}
end if
return RESULT

4 Phonological conditioning

This section first lists the phonological conditioning rules that have been implemented at the time of writing and which have been mentioned informally in, mainly [5] for locatives and the vowel-commencing verb roots, but not the others. They still seem incomplete and therefore also listed outside their algorithm environment. The algorithms are presented afterwards.

- vowel coalescence function (*phonoCondition* in the algorithms), where X and Y are the remainder of the word:
  - Xa + aY → XaY
  - Xa + (iY | eY) → XeY
  - (Xa, $X \neq$ ng) + uY → XoY // the '$X \neq$ ng' is an old remnant. as there's also nga + uY = ngo$Y$ now, so can be deleted. (or not?)
  - Xe + aY → XaY
  - Xe + iY → XeY
  - Xe + (oY | uY) → XoY
  - Xu + (aY | eY | iY | oY | uY) → XuY
  - nga + oY → ngo$Y$ //can this be generalised, as -a + o- = -o-?
  - nga + uY → ngo$Y$ //can be included in the 3rd one
- locative prefix (*phonoCondLocPrefix* in the algorithms), possibly still incomplete
  - if nc = 1a, 2a, 3a, or 17 → ku+*word* (subject to the vowel coalescence listed in the previous item)
  - for other ncs → e+*word* (subject to the vowel coalescence listed in the previous item)
- locative suffix (*phonoCondLocSuffix* in the algorithms)
  - regular cases:
    - Xa → Xeni
    - Xe → Xeni
    - Xi → Xini
    - Xo → Xweni
    - Xu + (¬ ph) → Xwini // the ph in second and third last position
    - Xu + ph → Xshini // the ph in second and third last position
    - otherwise *word*+ini
  - exceptions: imvilophu, idiphu, ifomu → word[0:-1] + ini.
- subject concord (*sc*) and vowel-commencing verb stem (*word*) and stem-minus-first-letter (X), named *phonoCondVerb* in the algorithms in the preceding sections:
  - (length(*sc*) ≥ 2, ¬ ku,lu) + (aX | eX) → [b/l/s/z]*word* // the remaining sc consonants (sc[:-1] to be more precise)
  - a + (aX | eX) → *word*
  - i + (aX | eX) → y*word*
  - u + (aX | eX) → w*word*
  - ku,lu + (aX | eX) → [k/l]w*word*
  - (length(*sc*) ≥ 2, ¬ ku) + oX → [b/l/s/z]*word* // the remaining sc consonants (sc[:-1] to be more precise)

- a,i + oX → *word*
- u + oX → w*word*
- ku + oX → kw*word*

- negative subject concord (*negsc*) and vowel-commencing verb stem (*word*), in the algorithm as *phonoCondNegSc*:
  - (aX | eX) → *negsc*y*word*
  - (iX | oX | uX) → *negsc*ng*word*

Also in this case, the algorithms were gradually extended in the code, so there may be some duplication (see also above) that may have yet to be refactored.

Algorithm 20 (VowelCoal) Vowel coalescence (or: two [the last letter of the first part and the first letter of the second part] becoming one)

---

input: two strings, *f*irst and *s*econd, respectively, where the former is to be agglutinated to the latter into a *n*ew word.
if $f_{[-1]}$ == 'a' **and** $s_{[0]}$ == 'a' **then**
  $n \leftarrow f_{[0:-1]}$a$s_{[1:]}$ {a+a = a}
else if $f_{[-1]}$ == 'a' **and** ($s_{[0]}$ == 'i' **or** $s_{[0]}$ == 'e') **then**
  $n \leftarrow f_{[0:-1]}$e$s_{[1:]}$ {a+i/e = e}
else if $f_{[-1]}$ == 'a' **and** f ! = 'nga' **and** $s_{[0]}$ == 'u' **then**
  $n \leftarrow f_{[0:-1]}$o$s_{[1:]}$ {a+u = o}
else if $f_{[-1]}$ == 'e' **and** $s_{[0]}$ == 'a' **then**
  $n \leftarrow f_{[0:-1]}$a$s_{[1:]}$ {e+a = a}
else if $f_{[-1]}$ == 'e' **and** $s_{[0]}$ == 'i' **then**
  $n \leftarrow f_{[0:-1]}$e$s_{[1:]}$ {e+i = e}
else if $f_{[-1]}$ == 'e' **and** ($s_{[0]}$ == 'o' **or** $s_{[0]}$ = 'u') **then**
  $n \leftarrow f_{[0:-1]}$o$s_{[1:]}$ {e+o/u = o}
else if $f_{[-1]}$ == 'u' **then**
  $n \leftarrow f s_{[1:]}$ {assuming the u is a 'stronger' vowel, for now}
else
  if f == 'nga' **and** $s_{[0]}$ == 'o' **then**
    $n \leftarrow$ ngo$s_{[1:]}$
  else if f == 'nga' **and** $s_{[0]}$ == 'u' **then**
    $n \leftarrow$ ngo$s_{[1:]}$
  else
    $n \leftarrow$ other {sentinel word to detect a phonological conditioning not covered yet}
  end if
end if
return n

Algorithm 21 (LocPre) Locative prefix for the noun or named entity.

input: word w and noun class nc
$l \leftarrow \mathsf{e}w$ {default case}
if $nc == 1\text{a}$ **or** $nc == 2\text{a}$ **or** $nc == 3a$ **or** $nc == 17$ **then**
    $l \leftarrow \mathit{VowelCoal}(\text{'ku'}, w)$
else
    $l \leftarrow \mathit{VowelCoal}(\text{'e'}, w)$
end if
return l

Algorithm 22 (LocSuf) Locative suffix for the noun or named entity.

input: word w
$\mathit{exceptions} = [\text{'imvilophu'}, \text{'idiphu'}, \text{'ifomu'}]$
if $w \in \mathit{exceptions}$ **or** $w_{[-1]} == \text{i}$ **then**
    $l \leftarrow w_{[0:-1]}\mathsf{ini}$ {note: 'ini' is the common case}
else if $w_{[-1]} == \text{a}$ **or** $w_{[-1]} == \text{e}$ **then**
    $l \leftarrow w_{[0:-1]}\mathsf{eni}$ {-a/-e = eni}
else if $w_{[-1]} == \text{o}$ **then**
    $l \leftarrow w_{[0:-1]}\mathsf{weni}$ {-o = weni}
else if $w_{[-1]} == \text{u}$ **and** $w_{[-3:-2]} \,!\!= \text{ph}$ **then**
    $l \leftarrow w_{[0:-1]}\mathsf{wini}$ {-u = wini}
else if $w_{[-1]} == \text{u}$ **and** $w_{[-3:-2]} == \text{ph}$ **then**
    $l \leftarrow w_{[0:-3]}\mathsf{shini}$ {-u = shini}
else
    $l \leftarrow w\mathsf{ini}$
end if
return l

Algorithm 23 (VowelVerb) Phonological conditioning for conjugation (SC with a vowel-commencing verb root).

```
input: word w and its noun class nc
sc ← getSC(nc)                                                  {get subject concord for that NC}
if (w[0] == 'a' or w[0] == 'e') and length(sc) ≥ 2 and sc != 'ku' and sc != 'lu' then
    conjv ← sc[:−1]w                                            {long sc + a-/e- = drop last letter of sc}
else if w[0] == 'a' or w[0] == 'e') and sc == 'a' then
    conjv ← w                                                   {a+a-/e- = drop sc}
else if (w[0] == 'a' or w[0] == 'e') and sc == 'i' then
    conjv ← yw                                                  {i+a-/e- = y+a-/e-}
else if (w[0] == 'a' or w[0] == 'e') and sc == 'u' then
    conjv ← ww                                                  {u+a-/e- = w+a-/e-}
else if (w[0] == 'a' or w[0] == 'e') and (sc == 'ku' or sc == 'lu') then
    conjv ← sc[0]ww                                             {ku/lu+a-/e- = k/l+w+a-/e-}
else if w[0] == 'o' and length(sc) ≥ 2 and sc != 'ku' then
    conjv ← sc[:−1]w                                            {long sc + o- = drop last letter of sc}
else if w[0] == 'o' and (sc == 'a' or sc == 'i') then
    conjv ← w                                                   {i/a + o- = o-}
else if w[0] == 'o' and sc == 'u' then
    conjv ← ww                                                  {u + o- = w + o-}
else if w[0] == 'o' and sc == 'ku' then
    conjv ← kww                                                 {ku + o- = kw + o-}
else
    conjv ← w                                                   {or: don't do anything}
end if
return conjv
```

Algorithm 24 (NegVowelVerb) Phonological conditioning for negated conjugation (NEG SC with a vowel-commencing verb root).

```
input: word w and its noun class nc
nsc ← getNEGSC(nc)                                  {get negative subject concord for that NC}
if w[0] == 'a' or w[0] == 'e' then
    negconjv ← nscyw                                {anything + a-/e- = anything + y + a-/e-}
else
    negconjv ← nscngw                               {anything + i-/o-/u- = anything + ng + i-/o-/u-}
end if
return negconjv
```

5 Architecture of the verbaliser

The OWL verbaliser is described in [7]. The architecture is depicted in Figure 2 and is such that one can:

- Run the Python code in the interpreter, feeding it just the strings in the format given by the definitions in the code;
- Use Owlready to fetch the vocabulary from an ontology stored in OWL/XML format, where the output is written to stdout/console;
- Use Owlready and Tkinter to fetch the vocabulary from an ontology stored in OWL/XML format *and* get pretty printing in colour.

To use the software, see the `readme.txt` in the zipfile for instructions.

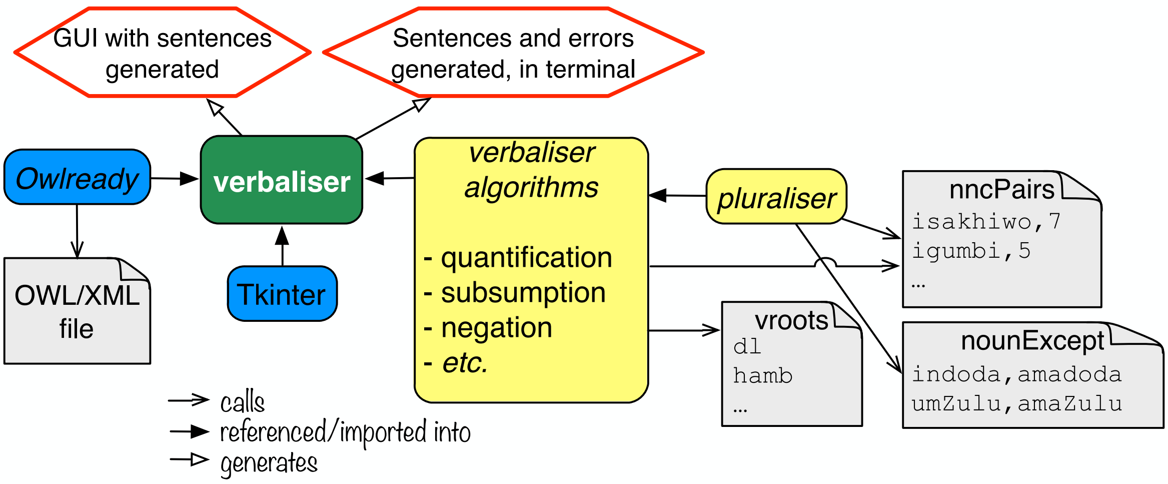

Figure 2: Principal components of the OWL verbaliser. (Source: [7])

Acknowledgements

As can be seen for the references, the main collaborator in trying to structure the linguistic knowledge is Langa Khumalo, and I hereby thank him for the collaboration.

This work is based on the research supported in part by the National Research Foundation of South Africa (CMK: Grant Number 93397).

References

[1] J. Byamugisha, C. M. Keet, and L. Khumalo. Pluralising nouns in isiZulu and similar languages. In A. Gelbukh, editor, *Proceedings of CICLing'16*, volume 9623 of *LNCS*, pages 271–283. Springer, 2018.

[2] C. Keet and L. Khumalo. Basics for a grammar engine to verbalize logical theories in isiZulu. In A. Bikakis et al., editors, *Proceedings of the 8th International Web Rule Symposium (RuleML'14)*, volume 8620 of *LNCS*, pages 216–225. Springer, 2014. August 18-20, 2014, Prague, Czech Republic.

[3] C. Keet and L. Khumalo. Toward verbalizing logical theories in isiZulu. In B. Davis, T. Kuhn, and K. Kaljurand, editors, *Proceedings of the 4th Workshop on Controlled Natural Language (CNL'14)*, volume 8625 of *LNAI*, pages 78–89. Springer, 2014. 20-22 August 2014, Galway, Ireland.

[4] C. M. Keet. Representing and aligning similar relations: parts and wholes in isizulu vs english. In J. Gracia, F. Bond, J. McCrae, P. Buitelaar, C. Chiarcos, and S. Hellmann, editors, *Language, Data, and Knowledge 2017 (LDK'17)*, volume 10318 of *LNAI*, pages 58–73. Springer, 2017. 19-20 June, 2017, Galway, Ireland.

[5] C. M. Keet and L. Khumalo. On the verbalization patterns of part-whole relations in isizulu. In *9th International Natural Language Generation conference (INLG'16)*, pages 174–183. ACL, 2016. 5-8 September, 2016, Edinburgh, UK.

[6] C. M. Keet and L. Khumalo. Toward a knowledge-to-text controlled natural language of isiZulu. *Language Resources and Evaluation*, 51(1):131–157, 2017.

[7] C. M. Keet, M. Xakaza, and L. Khumalo. Verbalising owl ontologies in isizulu with python. In E. Blomqvist, K. Hose, H. Paulheim, A. Lawrynowicz, F. Ciravegna, and O. Hartig, editors, *The Semantic Web: ESWC 2017 Satellite Events*, volume 10577 of *LNCS*, pages 59–64. Springer, 2017. 30 May - 1 June 2017, Portoroz, Slovenia.