A formalisation of the Cocke-Younger-Kasami algorithm

Maksym Bortin

May 4, 2022

Abstract

The theory provides a formalisation of the Cocke-Younger-Kasami algorithm [1] (CYK for short), an approach to solving the word problem for context-free languages. CYK decides if a word is in the languages generated by a context-free grammar in Chomsky normal form. The formalized algorithm is executable.

Contents

theory *CYK*
imports *Main*
begin

The theory is structured as follows. First section deals with modelling of grammars, derivations, and the language semantics. Then the basic properties are proved. Further, CYK is abstractly specified and its underlying recursive relationship proved. The final section contains a prototypical implementation accompanied by a proof of its correctness.

1 Basic modelling

1.1 Grammars in Chomsky normal form

A grammar in Chomsky normal form is here simply modelled by a list of production rules (the type CNG below), each having a non-terminal symbol on the lhs and either two non-terminals or one terminal symbol on the rhs.

datatype $('n, 't)$ *RHS* = *Branch* $'n$ $'n$
| *Leaf* $'t$

type-synonym $('n, 't)$ *CNG* = $('n \times ('n, 't)$ *RHS*$)$ *list*

Abbreviating the list append symbol for better readability

abbreviation *list-append* :: $'a$ *list* $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'a$ *list* (**infixr** $\cdot$ 65)
where $xs \cdot ys \equiv xs$ @ ys

1.2 Derivation by grammars

A *word form* (or sentential form) may be built of both non-terminal and terminal symbols, as opposed to a *word* that contains only terminals. By the usage of disjoint union, non-terminals are injected into a word form by *Inl* whereas terminals – by *Inr*.

type-synonym $('n, 't)$ *word-form* = $('n + 't)$ *list*
type-synonym $'t$ *word* = $'t$ *list*

A single step derivation relation on word forms is induced by a grammar in the standard way, replacing a non-terminal within a word form in accordance to the production rules.

definition *DSTEP* :: $('n, 't)$ *CNG* $\Rightarrow$ $(('n, 't)$ *word-form* $\times$ $('n, 't)$ *word-form*$)$ *set*
where *DSTEP G* = $\{(l \cdot [Inl\ N] \cdot r, x) \mid l\ N\ r\ rhs\ x.\ (N, rhs) \in set\ G \wedge$
(*case rhs of*
Branch A B $\Rightarrow x = l \cdot [Inl\ A, Inl\ B] \cdot r$
| *Leaf t* $\Rightarrow x = l \cdot [Inr\ t] \cdot r)\}$

abbreviation *DSTEP'* :: $('n, 't)$ *word-form* $\Rightarrow$ $('n, 't)$ *CNG* $\Rightarrow$ $('n, 't)$ *word-form* $\Rightarrow$ *bool* (- $-\!\!\rightarrow$ - [60, 61, 60] 61)
where $w\ -G\rightarrow\ w' \equiv (w, w') \in DSTEP\ G$

abbreviation *DSTEP-reflc* :: ('n, 't) *word-form* ⇒ ('n, 't) *CNG* ⇒ ('n, 't) *word-form* ⇒ *bool* (- −-→$^=$ - [60, 61, 60] 61)
where $w\ -G\rightarrow^=\ w' \equiv (w,\ w') \in (DSTEP\ G)^=$

abbreviation *DSTEP-transc* :: ('n, 't) *word-form* ⇒ ('n, 't) *CNG* ⇒ ('n, 't) *word-form* ⇒ *bool* (- −-→$^+$ - [60, 61, 60] 61)
where $w\ -G\rightarrow^+\ w' \equiv (w,\ w') \in (DSTEP\ G)^+$

abbreviation *DSTEP-rtransc* :: ('n, 't) *word-form* ⇒ ('n, 't) *CNG* ⇒ ('n, 't) *word-form* ⇒ *bool* (- −-→* - [60, 61, 60] 61)
where $w\ -G\rightarrow^*\ w' \equiv (w,\ w') \in (DSTEP\ G)^*$

1.3 The generated language semantics

The language generated by a grammar from a non-terminal symbol comprises all words that can be derived from the non-terminal in one or more steps. Notice that by the presented grammar modelling, languages containing the empty word cannot be generated. Hence in rare situations when such languages are required, the empty word case should be treated separately.

definition *Lang* :: ('n, 't) *CNG* ⇒ 'n ⇒ 't *word set*
where *Lang G S* = {*w*. [*Inl S*] $-G\rightarrow^+$ *map Inr w* }

So, for instance, a grammar generating the language $a^n b^n$ from the non-terminal ''*S*'' might look as follows.

definition *G-anbn* =
[(''*S*'', *Branch* ''*A*'' ''*T*''),
(''*S*'', *Branch* ''*A*'' ''*B*''),
(''*T*'', *Branch* ''*S*'' ''*B*''),
(''*A*'', *Leaf* ''*a*''),
(''*B*'', *Leaf* ''*b*'')]

Now the term *Lang G-anbn* ''*S*'' denotes the set of words of the form $a^n b^n$ with $n > 0$. This is intuitively clear, but not straight forward to show, and a lengthy proof for that is out of scope.

2 Basic properties

lemma *prod-into-DSTEP1* :
(*S*, *Branch A B*) ∈ *set G* ⟹
L · [*Inl S*] · *R* $-G\rightarrow$ *L* · [*Inl A*, *Inl B*] · *R*
⟨*proof*⟩

lemma *prod-into-DSTEP2* :
(*S*, *Leaf a*) ∈ *set G* ⟹
L · [*Inl S*] · *R* $-G\rightarrow$ *L* · [*Inr a*] · *R*

⟨*proof*⟩

lemma *DSTEP-D* :
$s\ -G\rightarrow\ t \Longrightarrow$
$\exists\, L\ N\ R\ rhs.\ s = L \cdot [Inl\ N] \cdot R \wedge (N,\ rhs) \in set\ G \wedge$
$(\forall\, A\ B.\ rhs = Branch\ A\ B \longrightarrow t = L \cdot [Inl\ A,\ Inl\ B] \cdot R) \wedge$
$(\forall\, x.\ rhs = Leaf\ x \longrightarrow t = L \cdot [Inr\ x] \cdot R)$
⟨*proof*⟩

lemma *DSTEP-append* :
assumes *a*: $s\ -G\rightarrow\ t$
shows $L\ \cdot\ s\ \cdot\ R\ -G\rightarrow\ L\ \cdot\ t\ \cdot\ R$
⟨*proof*⟩

lemma *DSTEP-star-mono* :
$s\ -G\rightarrow^*\ t \Longrightarrow length\ s \leq length\ t$
⟨*proof*⟩

lemma *DSTEP-comp* :
assumes *a*: $l \cdot r\ -G\rightarrow\ t$
shows $\exists\, l'\ r'.\ l\ -G\rightarrow^=\ l' \wedge r\ -G\rightarrow^=\ r' \wedge t = l' \cdot r'$
⟨*proof*⟩

theorem *DSTEP-star-comp1* :
assumes *A*: $l \cdot r\ -G\rightarrow^*\ t$
shows $\exists\, l'\ r'.\ l\ -G\rightarrow^*\ l' \wedge r\ -G\rightarrow^*\ r' \wedge t = l' \cdot r'$
⟨*proof*⟩

theorem *DSTEP-star-comp2* :
assumes *A*: $l\ -G\rightarrow^*\ l'$
and *B*: $r\ -G\rightarrow^*\ r'$
shows $l \cdot r\ -G\rightarrow^*\ l' \cdot r'$
⟨*proof*⟩

lemma *DSTEP-trancl-term* :

assumes *A*: $[Inl\ S] -G\rightarrow^{+} t$
and *B*: $Inr\ x \in set\ t$
shows $\exists N.\ (N, Leaf\ x) \in set\ G$
⟨*proof*⟩

2.1 Properties of generated languages

lemma *Lang-no-Nil* :
$w \in Lang\ G\ S \Longrightarrow w \neq []$
⟨*proof*⟩

lemma *Lang-rtrancl-eq* :
$(w \in Lang\ G\ S) = [Inl\ S] -G\rightarrow^{*} map\ Inr\ w$ (**is** $?L = (?p \in ?R^{*})$)
⟨*proof*⟩

lemma *Lang-term* :
$w \in Lang\ G\ S \Longrightarrow$
$\forall x \in set\ w.\ \exists N.\ (N, Leaf\ x) \in set\ G$
⟨*proof*⟩

lemma *Lang-eq1* :
$([x] \in Lang\ G\ S) = ((S, Leaf\ x) \in set\ G)$
⟨*proof*⟩

theorem *Lang-eq2* :
$(w \in Lang\ G\ S \wedge 1 < length\ w) =$
$(\exists A\ B.\ (S, Branch\ A\ B) \in set\ G \wedge (\exists l\ r.\ w = l \cdot r \wedge l \in Lang\ G\ A \wedge r \in Lang\ G\ B))$
(**is** $?L = ?R$)
⟨*proof*⟩

3 Abstract specification of CYK

A subword of a word w, starting at the position i (first element is at the position 0) and having the length j, is defined as follows.

definition $subword\ w\ i\ j = take\ j\ (drop\ i\ w)$

Thus, to any subword of the given word w CYK assigns all non-terminals from which this subword is derivable by the grammar G.

definition $CYK\ G\ w\ i\ j = \{S.\ subword\ w\ i\ j \in Lang\ G\ S\}$

3.1 Properties of *subword*

lemma *subword-length* :
$i + j \leq length\ w \Longrightarrow length(subword\ w\ i\ j) = j$
⟨*proof*⟩

lemma *subword-nth1* :
$i + j \leq length\ w \Longrightarrow k < j \Longrightarrow$
$(subword\ w\ i\ j)!k = w!(i + k)$
⟨*proof*⟩

lemma *subword-nth2* :
assumes *A*: $i + 1 \leq length\ w$
shows $subword\ w\ i\ 1 = [w!i]$
⟨*proof*⟩

lemma *subword-self* :
$subword\ w\ 0\ (length\ w) = w$
⟨*proof*⟩

lemma *take-split*[*rule-format*] :
$\forall\ n\ m.\ n \leq length\ xs \longrightarrow n \leq m \longrightarrow$
$take\ n\ xs \cdot take\ (m - n)\ (drop\ n\ xs) = take\ m\ xs$
⟨*proof*⟩

lemma *subword-split* :
$i + j \leq length\ w \Longrightarrow 0 < k \Longrightarrow k < j \Longrightarrow$
$subword\ w\ i\ j = subword\ w\ i\ k \cdot subword\ w\ (i + k)\ (j - k)$
⟨*proof*⟩

lemma *subword-split2* :
assumes *A*: $subword\ w\ i\ j = l \cdot r$
and *B*: $i + j \leq length\ w$
and *C*: $0 < length\ l$
and *D*: $0 < length\ r$
shows $l = subword\ w\ i\ (length\ l) \wedge r = subword\ w\ (i + length\ l)\ (j - length\ l)$
⟨*proof*⟩

3.2 Properties of *CYK*

lemma *CYK-Lang* :
$(S \in CYK\ G\ w\ 0\ (length\ w)) = (w \in Lang\ G\ S)$
⟨*proof*⟩

lemma *CYK-eq1* :
$i + 1 \leq length\ w \Longrightarrow$
$CYK\ G\ w\ i\ 1 = \{S.\ (S,\ Leaf\ (w!i)) \in set\ G\}$
⟨*proof*⟩

theorem *CYK-eq2* :
assumes *A*: $i + j \leq length\ w$
and *B*: $1 < j$
shows $CYK\ G\ w\ i\ j = \{X \mid X\ A\ B\ k.\ (X,\ Branch\ A\ B) \in set\ G \wedge A \in CYK\ G\ w\ i\ k \wedge B \in CYK\ G\ w\ (i + k)\ (j - k) \wedge 1 \leq k \wedge k < j\}$
⟨*proof*⟩

4 Implementation

One of the particularly interesting features of CYK implementation is that it follows the principles of dynamic programming, constructing a table of solutions for sub-problems in the bottom-up style reusing already stored results.

4.1 Main cycle

This is an auxiliary implementation of the membership test on lists.

fun *mem* :: $'a \Rightarrow 'a\ list \Rightarrow bool$
where
$mem\ a\ [] = False\ |$
$mem\ a\ (x\#xs) = (x = a \vee mem\ a\ xs)$

lemma *mem*[*simp*] :
$mem\ x\ xs = (x \in set\ xs)$
⟨*proof*⟩

The purpose of the following is to collect non-terminals that appear on the lhs of a production such that the first non-terminal on its rhs appears in the first of two given lists and the second non-terminal – in the second list.

fun *match-prods* :: $('n,\ 't)\ CNG \Rightarrow 'n\ list \Rightarrow 'n\ list \Rightarrow 'n\ list$
where $match\text{-}prods\ []\ ls\ rs = []\ |$
$match\text{-}prods\ ((X,\ Branch\ A\ B)\#ps)\ ls\ rs =$
$(if\ mem\ A\ ls \wedge mem\ B\ rs\ then\ X\ \#\ match\text{-}prods\ ps\ ls\ rs$
$else\ match\text{-}prods\ ps\ ls\ rs)\ |$

match-prods ((*X*, *Leaf a*)#*ps*) *ls rs* = *match-prods ps ls rs*

lemma *match-prods* :
(*X* ∈ *set*(*match-prods G ls rs*)) =
(∃ *A* ∈ *set ls*. ∃ *B* ∈ *set rs*. (*X*, *Branch A B*) ∈ *set G*)
⟨*proof*⟩

The following function is the inner cycle of the algorithm. The parameters i and j identify a subword starting at i with the length j, whereas k is used to iterate through its splits (which are of course subwords as well) all having the length greater 0 but less than j. The parameter T represents a table containing CYK solutions for those splits.

function *inner* :: ('*n*, '*t*) *CNG* ⇒ (*nat* × *nat* ⇒ '*n list*) ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ '*n list*
where *inner G T i k j* =
(*if k* < *j then match-prods G* (*T*(*i*, *k*)) (*T*(*i* + *k*, *j* − *k*)) @ *inner G T i* (*k* + *1*) *j*
else [])
⟨*proof*⟩
termination
⟨*proof*⟩

declare *inner.simps*[*simp del*]

lemma *inner* :
(*X* ∈ *set*(*inner G T i k j*)) =
(∃ *l*. *k* ≤ *l* ∧ *l* < *j* ∧ *X* ∈ *set*(*match-prods G* (*T*(*i*, *l*)) (*T*(*i* + *l*, *j* − *l*))))
(**is** *?L G T i k j* = *?R G T i k j*)
⟨*proof*⟩

Now the main part of the algorithm just iterates through all subwords up to the given length *len*, calls *inner* on these, and stores the results in the table T. The length j is supposed to be greater than 1 – the subwords of length 1 will be handled in the initialisation phase below.

function *main* :: ('*n*, '*t*) *CNG* ⇒ (*nat* × *nat* ⇒ '*n list*) ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ (*nat* × *nat* ⇒ '*n list*)
where *main G T len i j* = (*let T'* = *T*((*i*, *j*) := *inner G T i 1 j*) *in*
if i + *j* < *len then main G T' len* (*i* + *1*) *j*
else if j < *len then main G T' len 0* (*j* + *1*)
else T')
⟨*proof*⟩
termination
⟨*proof*⟩

declare *main.simps*[*simp del*]

lemma *main* :
assumes $1 < j$
and $i + j \leq$ *length* w
and $\bigwedge i'\ j'.\ j' < j \Longrightarrow 1 \leq j' \Longrightarrow i' + j' \leq$ *length* $w \Longrightarrow$ *set*$(T(i', j')) =$ *CYK* $G\ w\ i'\ j'$
and $\bigwedge i'.\ i' < i \Longrightarrow i' + j \leq$ *length* $w \Longrightarrow$ *set*$(T(i', j)) =$ *CYK* $G\ w\ i'\ j$
and $1 \leq j'$
and $i' + j' \leq$ *length* w
shows *set*((*main* $G\ T$ (*length* w) $i\ j)(i', j')) =$ *CYK* $G\ w\ i'\ j'$
⟨*proof*⟩

4.2 Initialisation phase

Similarly to *match-prods* above, here we collect non-terminals from which the given terminal symbol can be derived.

fun *init-match* :: $('n, 't)$ *CNG* $\Rightarrow$ $'t \Rightarrow$ $'n$ *list*
where *init-match* [] t = [] |
init-match $((X,$ *Branch* $A\ B)\#ps)\ t =$ *init-match* $ps\ t$ |
init-match $((X,$ *Leaf* $a)\#ps)\ t =$ (*if* $a = t$ *then* $X\ \#$ *init-match* $ps\ t$
else init-match $ps\ t$)

lemma *init-match* :
$(X \in$ *set*(*init-match* $G\ a)) =$
$((X,$ *Leaf* $a) \in$ *set* $G)$
⟨*proof*⟩

Representing the empty table.

definition *emptyT* = $(\lambda(i, j).$ [])

The following function initialises the empty table for subwords of length 1, i.e. each symbol occurring in the given word.

fun *init'* :: $('n, 't)$ *CNG* $\Rightarrow$ $'t$ *list* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\times$ *nat* $\Rightarrow$ $'n$ *list*
where *init'* G [] k = *emptyT* |
init' $G\ (t\#ts)\ k = (init'\ G\ ts\ (k + 1))((k, 1) :=$ *init-match* $G\ t)$

lemma *init'* :
assumes $i + 1 \leq$ *length* w
shows *set*(*init'* $G\ w\ 0\ (i, 1)) =$ *CYK* $G\ w\ i\ 1$
⟨*proof*⟩

The next version of initialization refines *init'* in that it takes additional account of the cases when the given word is empty or contains a terminal symbol that does not have any matching production (that is, *init-match* is an empty list). No initial table is then needed as such words can immediately be rejected.

```
fun init :: ('n, 't) CNG ⇒ 't list ⇒ nat ⇒ (nat × nat ⇒ 'n list) option
where init G [] k = None |
      init G [t] k = (case (init-match G t) of
                        [] ⇒ None
                      | xs ⇒ Some(emptyT((k, 1) := xs))) |
      init G (t#ts) k = (case (init-match G t) of
                           [] ⇒ None
                         | xs ⇒ (case (init G ts (k + 1)) of
                                   None ⇒ None
                                 | Some T ⇒ Some(T((k, 1) := xs))))
```

```
lemma init1:
  ‹init' G w k = T› if ‹init G w k = Some T›
  ⟨proof⟩
```

```
lemma init2 :
(init G w k = None) =
 (w = [] ∨ (∃ a ∈ set w. init-match G a = []))
⟨proof⟩
```

4.3 The overall procedure

```
definition cyk G S w = (case init G w 0 of
                          None ⇒ False
                        | Some T ⇒ let len = length w in
                                   if len = 1 then mem S (T(0, 1))
                                   else let T' = main G T len 0 2 in
                                        mem S (T'(0, len)))
```

```
theorem cyk :
cyk G S w = (w ∈ Lang G S)
⟨proof⟩
```

```
value [code]
  let G = [(0::int, Branch 1 2), (0, Branch 2 3),
           (1, Branch 2 1), (1, Leaf ''a''),
           (2, Branch 3 3), (2, Leaf ''b''),
           (3, Branch 1 2), (3, Leaf ''a'')]
  in map (cyk G 0)
     [[''b'',''a'',''a'',''b'',''a''],
      [''b'',''a'',''b'',''a'']]
```

end

References

[1] D. H. Younger. Recognition and parsing of context-free languages in time n3. *Information and Control*, 10(2):189 – 208, 1967.