A formalisation of the Cocke-Younger-Kasami
algorithm

Maksym Bortin

April 19, 2020

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

1 Basic modelling 2
  1.1 Grammars in Chomsky normal form ............... 2
  1.2 Derivation by grammars .......................... 2
  1.3 The generated language semantics ................. 3

2 Basic properties 3
  2.1 Properties of generated languages ............... 5

3 Abstract specification of CYK 5
  3.1 Properties of subword .......................... 6
  3.2 Properties of CYK .............................. 7

4 Implementation 7
  4.1 Main cycle .................................... 7
  4.2 Initialisation phase ............................ 9
  4.3 The overall procedure .......................... 10

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
```

Abbreviating the list append symbol for better readability

```
abbreviation list-append :: 'a list ⇒ 'a list ⇒ 'a list (infixr 65)
where xs · ys ≡ 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 \textit{Inl} whereas terminals – by \textit{Inr}.

```
type-synonym ('n, 't) word-form = ('n + '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 ⇒ (('n, 't) word-form × ('n, 't) word-form) set
where DSTEP G = \{(l · [Inl N] · r, x) | l N r rhs x, (N, rhs) ∈ set G ∧
                      \text{case rhs of}
                      Branch A B ⇒ x = l · [Inl A, Inl B] · r
                      | Leaf t ⇒ x = l · [Inr t] · r\}
```

Abbreviation

```
abbreviation DSTEP' :: ('n, 't) word-form ⇒ ('n, 't) CNG ⇒ ('n, 't) word-form
⇒ bool (- -→ - [60, 61] 61)
where w · G→ w' ≡ (w, w') ∈ DSTEP G
```

2
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** $\text{Lang} :: (',', 'n') \ CNG \Rightarrow 'n \Rightarrow 't$ word set

where $\text{Lang } G S = \{ w. \ [\text{Inl } S] - G \rightarrow^+ \text{ map Inr } w \}$

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

**Definition** $G-a^n b^n =

[(["S", \text{Branch } "A" "T"],
  ["S", \text{Branch } "A" "B"],
  ["T", \text{Branch } "S" "B"],
  ["A", \text{Leaf } "a"],
  ["B", \text{Leaf } "b"])]$

Now the term $\text{Lang } G-a^n b^n "S"$ 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 :

$(S, \text{Branch } A B) \in \text{set } G \Longrightarrow
L \cdot [\text{Inl } S] \cdot R - G \rightarrow L \cdot [\text{Inl } A, \text{Inl } B] \cdot R
\langle \text{proof} \rangle$

**Lemma** prod-into-DSTEP2 :

$(S, \text{Leaf } a) \in \text{set } G \Longrightarrow
L \cdot [\text{Inl } S] \cdot R - G \rightarrow L \cdot [\text{Inr } a] \cdot R$
lemma DSTEP-D:
\[ s \rightarrow G \rightarrow t \quad \Rightarrow \quad \exists \ L \ N \ R \ rhs. \ s = L \cdot [Inl \ N] \cdot R \land (N, \ rhs) \in \text{set } G \land \]
\[ (\forall \ A \ B. \ rhs = \text{Branch } A \ B \quad \rightarrow \quad t = L \cdot [Inl \ A, \ Inl \ B] \cdot R) \land \]
\[ (\forall \ x. \ rhs = \text{Leaf } x \quad \rightarrow \quad t = L \cdot [Inr \ x] \cdot R) \]

lemma DSTEP-append:
\(\text{assumes } a: s \rightarrow G \rightarrow t\)
\(\text{shows } L \cdot s \cdot R \rightarrow G \rightarrow L \cdot t \cdot R\)

lemma DSTEP-star-mono:
\(s \rightarrow G \rightarrow^* t \quad \Rightarrow \quad \text{length } s \leq \text{length } t\)

lemma DSTEP-comp:
\(\text{assumes } a: l \cdot r \rightarrow G \rightarrow t\)
\(\text{shows } \exists \ l' \ r'. \ l \rightarrow G \rightarrow^* l' \land r \rightarrow G \rightarrow^* r' \land t = l' \cdot r'\)

theorem DSTEP-star-comp1:
\(\text{assumes } A: l \cdot r \rightarrow G \rightarrow^* t\)
\(\text{shows } \exists \ l' \ r'. \ l \rightarrow G \rightarrow^* l' \land r \rightarrow G \rightarrow^* r' \land t = l' \cdot r'\)

theorem DSTEP-star-comp2:
\(\text{assumes } A: l \rightarrow G \rightarrow^* l'\)
\(\text{and } B: r \rightarrow G \rightarrow^* r'\)
\(\text{shows } l \cdot r \rightarrow G \rightarrow^* l' \cdot r'\)

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

2.1 Properties of generated languages

**lemma** Lang-no-Nil :

$w \in \text{Lang } G S \implies w \not= []$
(proof)

**lemma** Lang-rtrancl-eq :

$(w \in \text{Lang } G S) = [\text{Inl } S] \rightarrow G \rightarrow^* \text{map } \text{Inr } w$
(is $?L = (?p \in $?R*))
(proof)

**lemma** Lang-term :

$w \in \text{Lang } G S \implies$

$\forall x \in \text{set } w. \exists N. (N, \text{Leaf } x) \in \text{set } G$
(proof)

**lemma** Lang-eq1 :

$([x] \in \text{Lang } G S) = ((S, \text{Leaf } x) \in \text{set } G)$
(proof)

**theorem** Lang-eq2 :

$(w \in \text{Lang } G S \land 1 < \text{length } w) =$

$(\exists A B. (S, \text{Branch } A B) \in \text{set } G \land (\exists l r. w = l \cdot r \land l \in \text{Lang } G A \land r \in \text{Lang } G B))$
(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 $w i j = \text{take } j (\text{drop } i w)$

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

5
definition CYK G w i j = \{S. subword w i j \in \text{Lang } G S\}

3.1 Properties of subword

lemma subword-length :
\[i + j \leq \text{length } w \implies \text{length}(\text{subword } w i j) = j\]
⟨proof⟩

lemma subword-nth1 :
\[i + j \leq \text{length } w \implies k < j \implies (\text{subword } w i j)!k = w!(i + k)\]
⟨proof⟩

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

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

lemma take-split[rule-format] :
\[\forall n, m. \ n \leq \text{length } xs \implies n \leq m \implies \text{take } n \text{ xs} \cdot \text{take } (m - n) \ (\text{drop } n \text{ xs}) = \text{take } m \text{ xs}\]
⟨proof⟩

lemma subword-split :
\[i + j \leq \text{length } w \implies 0 < k \implies k < j \implies \text{subword } w i j = \text{subword } w i k \cdot \text{subword } w (i + k) (j - k)\]
⟨proof⟩

lemma subword-split2 :
assumes A: \[\text{subword } w i j = l \cdot r\]
and B: \[i + j \leq \text{length } w\]
and C: \[0 < \text{length } l\]
and D: \[0 < \text{length } r\]
shows \[l = \text{subword } w i (\text{length } l) \wedge r = \text{subword } w (i + \text{length } l) (j - \text{length } l)\]
⟨proof⟩
3.2 Properties of CYK

**Lemma CYK-Lang**:
\[(S \in CYK G w 0 (\text{length } w)) = (w \in \text{Lang } G S)\]
(\text{proof})

**Lemma CYK-eq1**:
\[i + 1 \leq \text{length } w \implies CYK G w i 1 = \{S. (S, \text{Leaf } (w!i)) \in \text{set } G\}\]
(\text{proof})

**Theorem CYK-eq2**:
\[\text{assumes } A: i + j \leq \text{length } w \quad \text{and } B: 1 < j\]
\[\text{shows } CYK G w i j = \{X \mid X A B k. (X, \text{Branch } A B) \in \text{set } G \land A \in CYK G w i k \land B \in CYK G (i + k) (j - k) \land 1 \leq k \land k < j\}\]
(\text{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**
\[
\begin{align*}
\text{mem} & : 'a \Rightarrow 'a \text{ list } \Rightarrow \text{bool} \\
\text{where} & \\
\text{mem} a [] & = \text{False} \\
\text{mem} a (x#xs) & = (x = a \lor \text{mem} a xs)
\end{align*}
\]

**Lemma mem[simp]**:
\[\text{mem } x \text{ xs } = (x \in \text{set } xs)\]
(\text{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**
\[
\begin{align*}
\text{match-prods} & : ('n, 't) \text{ CNG } \Rightarrow 'n \text{ list } \Rightarrow 'n \text{ list } \Rightarrow 'n \text{ list} \\
\text{where} & \\
\text{match-prods} [] & = [] \\
\text{match-prods} ((X, \text{Branch } A B)#ps) & = \left(\begin{array}{l}
\text{if mem } A \text{ ls } \land \text{mem } B \text{ rs then } X \# \text{ match-prods } ps \text{ ls rs } \\
\text{else } \text{match-prods } ps \text{ ls rs } \end{array}\right)
\end{align*}
\]
match-prods \((X, \text{Leaf} a)\#ps\) ls rs = match-prods ps ls rs

**Lemma** match-prods :
\[
(X \in \text{set}(\text{match-prods} G \text{ ls rs})) = \\
(\exists A \in \text{set ls}. \exists B \in \text{set rs}. (X, \text{Branch} A B) \in \text{set} G)
\]

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 =
\[
(\text{if } k < j \text{ then matched-prods } G (T(i, k)) (T(i + k, j - k)) @ inner G T i (k + 1) j \text{ else } [])
\]

**Termination**

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

**Lemma** inner :
\[
(X \in \text{set}(\text{inner} G T i k j)) = \\
(\exists l. k \leq l \land l < j \land X \in \text{set}(\text{match-prods} G (T(i, l)) (T(i + l, j - l))))
\]
(is ?L G T i k j = ?R G T i k j)

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 \(T' = T((i, j) := \text{inner} G T i 1 j)\) in
\[
\text{if } i + j < \text{len} \text{ then main } G T' \text{ len } (i + 1) j \\
\text{else if } j < \text{len} \text{ then main } G T' \text{ len } 0 (j + 1) \\
\text{else } T')
\]

**Termination**

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

8
lemma main:
assumes $1 < j$
and $i + j \leq \text{length } w$
and $\bigwedge i' j'. j' < j \implies 1 \leq j' \implies i' + j' \leq \text{length } w \implies \text{set}(T(i', j')) = \text{CYK } G w i' j'$
and $1 \leq j' \implies i' + j' \leq \text{length } w \implies \text{set}(T(i', j)) = \text{CYK } G w i' j$
shows \text{set}((\text{main } G T (\text{length } w) i j)(i', j')) = \text{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 ⇒ 't ⇒ '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:
$(X \in \text{set}(\text{init-match } G a)) =
((X, \text{Leaf } a) \in \text{set } G)$
⟨proof⟩

Representing the empty table.

definition emptyT = (λ(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 ⇒ 't list ⇒ nat ⇒ nat × nat ⇒ '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 $i + 1 \leq \text{length } w$
shows \text{set}(init′ G w 0 (i, 1)) = \text{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 [rule-format] :
∀ T. init G w k = Some T →
    init’ G w k = 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