A formalisation of the Cocke-Younger-Kasami algorithm

Maksym Bortin

October 11, 2017

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 . . . . . . . . . . . . 9

3 Abstract specification of CYK 12
   3.1 Properties of subword . . . . . . . . . . . . . . . . . . . . . 12
   3.2 Properties of CYK . . . . . . . . . . . . . . . . . . . . . . . 13

4 Implementation 15
   4.1 Main cycle . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
   4.2 Initialisation phase . . . . . . . . . . . . . . . . . . . . . . . 21
   4.3 The overall procedure . . . . . . . . . . . . . . . . . . . . . 22
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.

\[
\text{datatype } (\text{'n, 't) RHS = Branch \, 'n \, 'n | Leaf \, 't}
\]

\[
\text{type-synonym } (\text{'n, 't) CNG = (\text{'n × ('n, 't) RHS) list}
\]

Abbreviating the list append symbol for better readability

\[
\text{abbreviation list-append :: 'a list ⇒ 'a list ⇒ 'a list (infixr \cdot 65)}
\]

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 \(Inr\).

\[
\text{type-synonym} (\text{'n, 't) word-form = (\text{'n + 't) list}
\]

\[
\text{type-synonym} \ 't word = 't list
\]

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.

\[
\text{definition DSTEP :: ('n, 't) CNG ⇒ ('n, 't) word-form × ('n, 't) word-form) set}
\]

\[
\text{where DSTEP G = \{(l \cdot [Inl N] \cdot r, x) | l \, N \, r \, x. (N, rhs) \in set G \wedge (case rhs of}
\]

\[
\text{Branch A B ⇒ x = l \cdot [Inl A, Inl B] \cdot r}
\]

\[
| Leaf t ⇒ x = l \cdot [Inr t] \cdot r\}}
\]

\[
\text{abbreviation DSTEP' :: ('n, 't) word-form ⇒ ('n, 't) CNG ⇒ ('n, 't) word-form}
\]

\[
\Rightarrow bool (-→ - [60, 61, 60] 61)
\]

\[
\text{where w \rightarrow G \rightarrow w' \equiv (w, w') \in DSTEP G}
\]
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 \( \text{Lang} \) :: \((n, t)\) CNG \(\Rightarrow\) \(\text{'}n\text{'} \Rightarrow \text{'}t\) 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 \(G-anbn = \)

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

Now the term \(\text{Lang} G-anbn "S"\) denotes the set of words of the form \(a^n b^n\) with \(n > 0\). This is intuitively clear, but not straightforward to show, and a lengthy proof for that is out of scope.

2 Basic properties

lemma \(\text{prod-into-DSTEP1} : \)

\((S, \text{Branch} A B) \in \text{set} G \implies\)

\(L \cdot [\text{Inl} S] \cdot R \rightarrow G \rightarrow L \cdot [\text{Inl} A, \text{Inl} B] \cdot R\)

by (simp add: DSTEP-def, rule-tac \(x=L \in\) \(\text{exI}\), force)

lemma \(\text{prod-into-DSTEP2} : \)

\((S, \text{Leaf} a) \in \text{set} G \implies\)

\(L \cdot [\text{Inl} S] \cdot R \rightarrow G \rightarrow L \cdot [\text{Inr} a] \cdot R\)
by(simp add: DSTEP-def, rule-tac x=L in exI, force)

lemma DSTEP-D:
\[ s \rightarrow^* t \quad \Rightarrow \quad \exists L N R \text{ rhs. } s = L \cdot [\text{Inl N}] \cdot R \land (N, \text{ rhs}) \in \text{set G} \land \]
\[ (\forall A B. \text{ rhs = Branch A B} \quad \rightarrow \quad t = L \cdot [\text{Inl A}, \text{Inl B}] \cdot R) \land \]
\[ (\forall x. \text{ rhs = Leaf x} \quad \rightarrow \quad t = L \cdot [\text{Inr x}] \cdot R) \]
by(unfold DSTEP-def, clarsimp, simp split: RHS.split-asm, blast+)

lemma DSTEP-append:
\[ \text{assumes a: s \rightarrow^* t} \]
\[ \text{shows L \cdot s \cdot R \rightarrow^* L \cdot t \cdot R} \]
proof -
  from a have \[ \exists l N r \text{ rhs. } s = l \cdot [\text{Inl N}] \cdot r \land (N, \text{ rhs}) \in \text{set G} \land \]
  \[ (\forall A B. \text{ rhs = Branch A B} \quad \rightarrow \quad t = l \cdot [\text{Inl A}, \text{Inl B}] \cdot r) \land \]
  \[ (\forall x. \text{ rhs = Leaf x} \quad \rightarrow \quad t = l \cdot [\text{Inr x}] \cdot r) \quad (\text{is } \exists l N r \text{ rhs. } ?P l N r \text{ rhs}) \]
  by(rule DSTEP-D)
  then obtain l N r \text{ rhs where } ?P l N r \text{ rhs by blast}
  thus ?thesis by
  (case-tac rhs, clarsimp+)
qed

lemma DSTEP-star-mono:
\[ s \rightarrow^* t \quad \Rightarrow \quad \text{length } s \leq \text{length } t \]
proof(erule rtrancl_induct, simp)
  fix t u
  assume s \rightarrow^* t
  assume a: t \rightarrow^* u
  assume b: \text{length } s \leq \text{length } t
  show \text{length } s \leq \text{length } u
  proof -
    from a have \[ \exists L N R \text{ rhs. } t = L \cdot [\text{Inl N}] \cdot R \land (N, \text{ rhs}) \in \text{set G} \land \]
    \[ (\forall A B. \text{ rhs = Branch A B} \quad \rightarrow \quad u = L \cdot [\text{Inl A}, \text{Inl B}] \cdot R) \land \]
    \[ (\forall x. \text{ rhs = Leaf x} \quad \rightarrow \quad u = L \cdot [\text{Inr x}] \cdot R) \quad (\text{is } \exists L N R \text{ rhs. } ?P L N R \text{ rhs}) \]
    by(rule DSTEP-D)
    then obtain L N R \text{ rhs where } ?P L N R \text{ rhs by blast}
    with b show ?thesis
    by(case-tac rhs, clarsimp+)
  qed

qed
qed

lemma $DSTEP\cdot\text{comp}$:
assumes $a:\ l\cdot r - G \rightarrow t$
shows $\exists l'\ r'. l - G \rightarrow\ l'\land r - G \rightarrow r'\land t = l'\cdot r'$
proof –
from $a$ have $\exists L\ N\ R\ rhs.\ l\cdot r = L\cdot [\text{Inl}\ N]\cdot R\land (N,\ rhs)\in\text{set}\ G\land$
($\forall A\ B.\ rhs = \text{Branch}\ A\ B\rightarrow t = L\cdot [\text{Inl}\ A,\ \text{Inl}\ B]\cdot R)\land$
($\forall x.\ rhs = \text{Leaf}\ x\rightarrow t = L\cdot [\text{Inr}\ x]\cdot R)$
(is $\exists L\ N\ R\ rhs.\ ?T$
$L\ N\ R\ rhs)$
by(rule $DSTEP\cdot D$)
then obtain $L\ N\ R\ rhs$ where $b: ?T\ L\ N\ R\ rhs$ by blast
hence $l\cdot r = L\cdot [\text{Inl}\ N]\cdot R$ by simp
hence $\exists u.\ (l = L\cdot u\land u\cdot r = \text{Inl}\ N\ #\ R)\lor (l\cdot u = L\land r = u\cdot \text{Inl}\ N\ #\ R)$
by(rule append-eq-append-conv2[THEN iffD1])
then obtain $xs$ where $c: l = L\cdot xs\land xs\cdot r = \text{Inl}\ N\ #\ R\lor l\cdot xs = L\land r = xs\cdot \text{Inl}\ N\ #\ R$ (is $?C1\lor ?C2)$ by blast
show $?thesis$
proof(cases $rhs$)
  case $\text{Leaf}\ x$
  with $b$ have $d: t = L\cdot [\text{Inr}\ x]\cdot R\land (N,\ \text{Leaf}\ x)\in\text{set}\ G$ by simp
  from $c$ show $?thesis$
  proof
    assume $e: ?C1$
    show $?thesis$
    proof
      case $\text{Nil}$ with $d$ and $e$ show $?thesis$
      by(clarsimp, rule-tac $x=L$ in $\text{exI}$, simp add: $DSTEP\cdot\text{def}$, simp split: RHS.split, blast)
      next
      case $\text{Cons}\ z\ zs$ with $d$ and $e$ show $?thesis$
      by(rule-tac $x=L$ in $\text{exI}$, clarsimp, simp add: $DSTEP\cdot\text{def}$, simp split: RHS.split, blast)
    qed
    next
    assume $e: ?C2$
    show $?thesis$
    proof(cases $xs$)
      case $\text{Nil}$ with $d$ and $e$ show $?thesis$
      by(rule-tac $x=L$ in $\text{exI}$, clarsimp, simp add: $DSTEP\cdot\text{def}$, simp split: RHS.split, blast)
      next
      case $\text{Cons}\ z\ zs$ with $d$ and $e$ show $?thesis$
      by(rule-tac $x=L$ in $\text{exI}$, clarsimp, simp add: $DSTEP\cdot\text{def}$, simp split: RHS.split, blast)
    qed
  qed
next
assume $e: ?C2$
show $?thesis$
proof(cases $xs$)
  case $\text{Nil}$ with $d$ and $e$ show $?thesis$
  by(rule-tac $x=L$ in $\text{exI}$, clarsimp, simp add: $DSTEP\cdot\text{def}$, simp split: RHS.split, blast)
next
  case $\text{Cons}\ z\ zs$ with $d$ and $e$ show $?thesis$
  by(rule-tac $x=L$ in $\text{exI}$, clarsimp, simp add: $DSTEP\cdot\text{def}$, simp split: RHS.split, blast)
rule-tac $x=z\#zs$ in $\text{exI}$, rule-tac $x=N$ in $\text{exI}$, rule-tac $x=R$ in $\text{exI}$, simp, rule-tac $x=\text{Leaf}\ x$ in $\text{exI}$, simp)
qed
qed

next

case (Branch A B)

with b have d: |t = L · [Inl A, Inl B] · R ∧ (N, Branch A B) ∈ set G | by simp

from c show ?thesis

proof

assume e: ?C1

show ?thesis

proof (cases xs)

  case Nil with d and e show ?thesis

  by (clarsimp, rule-tac x = L in exI, simp add: DSTEP-def, simp split: RHS.split, blast)

next

  case (Cons z zs) with d and e show ?thesis

  by (rule-tac x = L [Inl A, Inl B] · zs in exI, clarsimp, simp add: DSTEP-def, simp split: RHS.split, blast)

qed

next

assume e: ?C2

show ?thesis

proof (cases xs)

  case Nil with d and e show ?thesis

  by (clarsimp, rule-tac x = L in exI, simp add: DSTEP-def, simp split: RHS.split, blast)

next

  case (Cons z zs) with d and e show ?thesis

  by (rule-tac x = l in exI, clarsimp, simp split: RHS.split, blast)

  rule-tac x = z # zs in exI, rule-tac x = N in exI, rule-tac x = R in exI, simp, rule-tac x = Branch A B in exI, simp)

qed

qed

qed

 theorem DSTEP-star-comp1 :

assumes A: l · r − G→∗ t

shows ∃ l' r'. l − G→∗ l' ∧ r − G→∗ r' ∧ t = l' · r'

proof –

  have ∀ s. s − G→∗ t ≡⇒

    ∀ l r. s = l · r → (∃ l' r'. l − G→∗ l' ∧ r − G→∗ r' ∧ t = l' · r') (is ∀ s. ?P s t ≡⇒ ?Q s t)

  proof (erule trivancI-induct, force)

    fix s t u

    assume ?P s t

    assume a: t − G→ u
assume b: \( ?Q \ s \ t \)

show \( ?Q \ s \ u \)

proof (clarify)

fix \( l \ r \)

assume \( s = l \cdot r \)

with \( b \) have \( \exists l' r'. \ l - G \rightarrow^* \ l' \land r - G \rightarrow^* \ r' \land t = l' \cdot r' \) by simp

then obtain \( l' r' \) where \( c: \ l - G \rightarrow^* \ l' \land r - G \rightarrow^* \ r' \land t = l' \cdot r' \) by blast

with \( a \) have \( l' \cdot r' - G \rightarrow^* \ u \) by simp

hence \( \exists l'' r''. \ l' - G \rightarrow^* \ l'' \land r' - G \rightarrow^* \ r'' \land u = l'' \cdot r'' \) by (rule DSTEP-comp)

then obtain \( l'' r'' \) where \( l' - G \rightarrow^* \ l'' \land r' - G \rightarrow^* \ r'' \land u = l'' \cdot r'' \) by blast

with \( c \) show \( \exists l'' r''. \ l - G \rightarrow^* \ l'' \land r - G \rightarrow^* \ r'' \land u = l'' \cdot r'' \) by (rule-tac \( x=l'' \) in \( \exists l' r'. \ l - G \rightarrow^* \ l' \land r - G \rightarrow^* \ r' \land t = l' \cdot r' \) by simp)

qed

next

fix \( s \ t \)

assume \( a: \ s - G \rightarrow \ t \)

assume \( b: \ ?Q \ l \ s \)

show \( ?Q \ l \ t \)

proof (clarsimp)

fix \( r \ r' \)

assume \( r - G \rightarrow^* \ r' \)

with \( b \) have \( c: \ l \cdot r - G \rightarrow^* \ s \cdot r' \) by simp
from a have \( s \cdot r' \rightarrow_{G} t \cdot r' \) by (drule_tac \( L=[] \) and \( R=r' \) in DSTEP-append, simp)
  with c show \( l \cdot r \rightarrow_{G}^{*} t \cdot r' \) by simp
  qed
  qed
with A and B show \(?thesis\) by simp
qed

lemma DSTEP-trancl-term:
assumes A: \([\text{Inl } S] \rightarrow_{G} t\)
  and B: \(\text{Inr } x \in \text{set } t\)
shows \(\exists N. (N, \text{Leaf } x) \in \text{set } G\)
proof
  have \([\text{Inl } S] \rightarrow_{G}^{*} t \Rightarrow \forall x. \text{Inr } x \in \text{set } t \rightarrow (\exists N. (N, \text{Leaf } x) \in \text{set } G)\) (is \(?P t \Rightarrow \ ?Q t\))
  proof (erule trancl-induct)
    fix t
    assume a: \([\text{Inl } S] \rightarrow_{G} t\)
    show \(?Q t\)
      proof
        from a have \(\exists \text{rhs}. (S, \text{rhs}) \in \text{set } G \land (\forall A B. \text{rhs} = \text{Branch } A B \rightarrow t = [\text{Inl } A, \text{Inl } B]) \land (\forall x. \text{rhs} = \text{Leaf } x \rightarrow t = [\text{Inr } x])\) (is \(?P \text{rhs}\))
        by (simp add: DSTEP-def, clarsimp, simp split: RHS.split-asm, case-tac l, force, simp, clarsimp, simp split: RHS.split-asm, case-tac l, force, simp)
        then obtain rhs where \(?P \text{rhs}\) by blast
        thus \(?thesis\)
        by (case-tac rhs, clarsimp, force)
      qed
    qed
  next
    fix s t
    assume a: \(s \rightarrow_{G} t\)
    assume b: \(?Q s\)
    show \(?Q t\)
      proof
        from a have \(\exists L N R. \text{rhs} = L \cdot [\text{Inl } N] \cdot R \land (N, \text{rhs}) \in \text{set } G \land (\forall A B. \text{rhs} = \text{Branch } A B \rightarrow t = L \cdot [\text{Inl } A, \text{Inl } B] \cdot R) \land (\forall x. \text{rhs} = \text{Leaf } x \rightarrow t = L \cdot [\text{Inr } x] \cdot R)\) (is \(?P L N R \text{rhs}\))
        by (rule DSTEP-D)
        then obtain L N R rhs where \(?P L N R \text{rhs}\) by blast
        with b show \(?thesis\)
        by (case-tac rhs, clarsimp, force)
      qed
  qed
with A and B show \(?thesis\) by simp
2.1 Properties of generated languages

**Lemma Lang-no-Nil:**
\[ w \in \text{Lang } G \iff S \implies w \neq [] \]
by (simp add: Lang-def, drule trancl-into-rtrancl, drule DSTEP-star-mono, force)

**Lemma Lang-rtrancl-eq:**
\[ (w \in \text{Lang } G \iff S) = [\text{Inl } S] - G \implies \text{map } \text{Inr } w \]
(is \(L = (\exists p \in ?R^+))\)
proof (simp add: Lang-def, rule iffI, erule trancl-into-rtrancl)
assume \(p \in ?R^+\)
hence \(p \in (?R^+)^\circ\) by (subst rtrancl-trancl-refcl[THEN sym], assumption)
hence \([\text{Inl } S] = \text{map } \text{Inr } w \lor p \in ?R^+\) by force
thus \(p \in ?R^+\) by (case-tac \(w\), simp-all)
qed

**Lemma Lang-term:**
\[ w \in \text{Lang } G \iff S \implies \forall x \in \text{set } w. \exists N. (N, \text{Leaf } x) \in \text{set } G \]
by (clarsimp simp add: Lang-def, drule DSTEP-trancl-term, simp, erule imageI, assumption)

**Lemma Lang-eq1:**
\[ ([x] \in \text{Lang } G \iff S) = ((S, \text{Leaf } x) \in \text{set } G) \]
proof (simp add: Lang-def, rule iffI, subst (asm) trancl-unfold-left, clarsimp)
fix \(t\)
assume \(a: [\text{Inl } S] \to G \to t\)
assume \(b: t \to G \to [\text{Inr } x]\)
note DSTEP-star-mono[OF \(b\), simplified]
hence \(c: \text{length } \leq 1\) by simp
have \(\exists z. t = [z]\)
proof (cases \(t\))
assume \(t = []\)
with \(b\) have \(d: [] \to G \to [\text{Inr } x]\) by simp
have \(\wedge s, ([], s) \in (\text{DSTEP } G)^\circ \implies s = []\)
by (erule rtrancl-induct, simp-all, drule DSTEP-D, clarsimp)
note this[OF \(d\)]
thus \(?\)thesis by simp
next
fix \(z\) \(zs\)
assume \(t = z \# zs\)
with c show thesis by force
qed

with a have \( \exists z. (S, \text{Leaf } z) \in \text{set } G \land t = [\text{Inr } z] \)
by(clarsimp simp add: DSTEP-def, simp split: RHS.split-asm, case-tac l, simp-all)

with b show (S, \text{Leaf } x) \in \text{set } G
proof(clarsimp)
fix z
assume c: (S, \text{Leaf } z) \in \text{set } G
assume [Inr z] \rightarrow G \rightarrow+ [Inr x]
hence [(\text{Inr } z), (\text{Inr } x)] \in ((\text{DSTEP } G)^+)^= \text{ by simp}
hence [Inr z] = [\text{Inr } x] \lor [\text{Inr } z] \rightarrow G \rightarrow+ [\text{Inr } x] \text{ by force}
hence x = z
proof
assume [Inr z] = [\text{Inr } x] thus thesis by simp
next
assume [Inr z] = [\text{Inr } x]
hence \exists u. [\text{Inr } z] - G - u \land u - G - [\text{Inr } x] \text{ by subst (asm) trancl-unfold-left, force}
then obtain u where [\text{Inr } z] - G - u \text{ by blast}
thus thesis by (clarsimp simp add: DSTEP-def, case-tac l, simp-all)
qed

with c show thesis by simp
qed

next
assume a: (S, \text{Leaf } x) \in \text{set } G
show [\text{Inl } S] - G \rightarrow+ [\text{Inr } x]
by(rule r-into-trancl, simp add: DSTEP-def, rule-tac xs=[], in exl, rule-tac x=x in exl, simp, rule-tac x=Leaf x in exl, simp add: a)
qed

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(rule iffI, clarify, subst (asm) Lang-def, simp, subst (asm) trancl-unfold-left, clarsimp)
have map-Inr-split : \( \\forall xs. \forall zs w. \text{map } \text{Inr } w = xs \cdot zs \rightarrow \)
\( (\exists u \ v. \ w = u \cdot v \land xs = \text{map } \text{Inr } u \land zs = \text{map } \text{Inr } v) \)
by(induct-tac xs, simp, force)
fix t
assume a: Suc 0 < \text{length } w
assume b: [\text{Inl } S] - G - t
assume c: t - G \rightarrow+ \text{map } \text{Inr } w
from b have \( \exists A \ B. (S, \text{Branch } A \ B) \in \text{set } G \land t = [\text{Inl } A, \text{Inl } B] \)
proof\((\text{simp add: DSTEP-def, clarify, case-tac l, simp-all, simp split: RHS.split-asm, clarify})\)

\begin{align*}
\text{fix } x \\
\text{assume } t = [\text{Inr } x] \\
\text{with } c \text{ have } d: [\text{Inr } x] \rightarrow^* \text{ map } \text{Inr } w \text{ by simp} \\
\text{have } \forall x. [\text{Inr } x] \rightarrow^* s \implies s = [\text{Inr } x] \\
\text{by (erule rtrancl-induct, simp-all, drule DSTEP-D, clarsimp, case-tac L, simp-all)} \\
\text{note } \text{this[OF } d]\text{]} \\
\text{hence } w = [x] \text{ by (case-tac } w, \text{ simp-all)} \\
\text{with a show False by simp} \\
\text{qed}
\end{align*}

then obtain \(A B\) \text{ where } d: (S, \text{ Branch } A B) \in \text{ set } G \land t = [\text{Inl } A, \text{ Inl } B] \text{ by blast}

\begin{align*}
\text{with } c \text{ have } e: \text{[Inl } A] \cdot \text{[Inl } B] \rightarrow^* \text{ map } \text{Inr } w \text{ by simp} \\
\text{note DSTEP-star-comp1[OF } e]\text{]} \\
\text{then obtain } l' r' \text{ where } e: \text{[Inl } A] \rightarrow^* l' \land \text{[Inl } B] \rightarrow^* r' \land \\
\text{map } \text{Inr } w = l' \cdot r' \text{ by blast} \\
\text{note map-Inr-split[rule-format, OF } e]\text{[THEN conjunct2, THEN conjunct2]} \\
\text{then obtain } u v \text{ where } f: w = u \cdot v \land l' = \text{ map } \text{Inr } u \land r' = \text{ map } \text{Inr } v \text{ by blast} \\
\text{with } e \text{ have } g: \text{[Inl } A] \rightarrow^* \text{ map } \text{Inr } u \land \text{[Inl } B] \rightarrow^* \text{ map } \text{Inr } v \text{ by simp} \\
\text{show } \exists R \\
\text{by (rule-tac } x=A \text{ in } \text{exI}, \text{ rule-tac } x=B \text{ in } \text{exI}, \text{ simp add: } d, \\
\text{ rule-tac } x=u \text{ in } \text{exI}, \text{ rule-tac } x=v \text{ in } \text{exI}, \text{ simp add: } f, \\
\text{(subt } \text{Lang-rtrancl-eq)+, \text{ rule } g) \\
\text{next}
\end{align*}

assume \(\exists R\)

then obtain \(A B l r\) \text{ where } a: (S, \text{ Branch } A B) \in \text{ set } G \land w = l \cdot r \land l \in \text{Lang } G A \land r \in \text{Lang } G B \text{ by blast}

\begin{align*}
\text{have } [\text{Inl } A] \cdot [\text{Inl } B] \rightarrow^* \text{ map } \text{Inr } l \cdot \text{ map } \text{Inr } r \\
\text{by (rule DSTEP-star-comp2, subt } \text{Lang-rtrancl-eq[THEN sym], simp add: } a, \\
\text{ subt } \text{Lang-rtrancl-eq[THEN sym], simp add: } a) \\
\text{hence } b: [\text{Inl } A] \cdot [\text{Inl } B] \rightarrow^* \text{ map } \text{Inr } w \text{ by (simp add: } a) \\
\text{have } c: w \in \text{Lang } G S \\
\text{by (simp add: } \text{Lang-def, subt } \text{trancl-unfold-left, rule-tac } b=[\text{Inl } A] \cdot [\text{Inl } B] \text{ in } \text{reccomplI}, \\
\text{ simp add: } \text{DSTEP-def, rule-tac } x=[] \text{ in } \text{exI}, \text{ rule-tac } x=S \text{ in } \text{exI}, \text{ rule-tac } x=[] \text{ in } \text{exI}, \\
\text{ simp, rule-tac } x=\text{Branch } A B \text{ in } \text{exI}, \text{ simp add: } a[\text{THEN conjunct1}, \text{ rule } b) \\
\text{thus } \exists l \\
\text{proof}
\end{align*}

\begin{align*}
\text{show } l < \text{ length } w \\
\text{proof (simp add: } a, \text{ rule ccontr, drule leI) } \\
\text{assume } \text{length } l + \text{ length } r \leq \text{ Suc } 0 \\
\text{hence } l = [] \lor r = [] \text{ by (case-tac } l, \text{ simp-all)} \\
\text{thus False} \\
\text{proof}
\end{align*}

\begin{align*}
\text{assume } l = [] \\
\text{with } a \text{ have } [] \in \text{Lang } G A \text{ by simp} \\
\end{align*}
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 \( G \).

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

### 3.1 Properties of subword

**lemma** subword-length :
\[ i + j \leq \text{length } w \implies \text{length}(\text{subword } w \ i \ j) = j \]
by (simp add: subword-def)

**lemma** subword-nth1 :
\[ i + j \leq \text{length } w \implies k < j \implies (\text{subword } w \ i \ j)!k = w!(i + k) \]
by (simp add: subword-def)

**lemma** subword-nth2 :
**assumes** \( A: i + 1 \leq \text{length } w \)
**shows** subword \( w \ i \ 1 = [w!i] \)
**proof** –
**note** subword-length\([OF A]\]
**hence** \( \exists x. \ \text{subword } w \ i \ 1 = [x] \) by (case-tac subword \( w \ i \ 1 \), simp-all)
**then obtain** \( x \) **where** \( a:\text{subword } w \ i \ 1 = [x] \) **by blast**
**note** subword-nth1\([OF A, where k=(0 :: nat), simplified]\)
**with** \( a \) **have** \( x = w!i \) **by simp**
**with** \( a \) **show** ?thesis **by simp**
**qed**
lemma subword-self:
subword w 0 (length w) = w
by(simp add: subword-def)

lemma take-split[rule-format]:
∀ n m. n ≤ length xs → n ≤ m →
take n xs · take (m − n) (drop n xs) = take m xs
by(induct-tac xs, clarsimp+, case-tac n, simp-all, case-tac m, simp-all)

lemma subword-split:
i + j ≤ length w =⇒ 0 < k =⇒ k < j =⇒
subword w i j = subword w i k · subword w (i + k) (j − k)
by(simp add: subword-def, subst take-split[where n=k, THEN sym], simp-all,
rule-tac f=λx. take (j − k) (drop x w) in arg-cong, simp)

lemma subword-split2:
assumes A: subword w i j = l · r
and B: i + j ≤ length w
and C: 0 < length l
and D: 0 < length r
shows l = subword w i (length l) ∧ r = subword w (i + length l) (j − length l)
proof −
have a: length(subword w i j) = j by(rule subword-length, rule B)
note arg-cong[where f=length, OF A]
with a and D have b: length l < j by force
with B have c: i + length l ≤ length w by force
have subword w i j = subword w i (length l) · subword w (i + length l) (j − length l)
by(rule subword-split, rule B, rule C, rule b)
with A have d: l · r = subword w i (length l) · subword w (i + length l) (j − length l) by simp
show ?thesis
by(rule append-eq-append-conv[THEN iffD1], subst subword-length, rule c, simp, rule d)
qed

3.2 Properties of CYK

lemma CYK-Lang:
(S ∈ CYK G w 0 (length w)) = (w ∈ Lang G S)
by(simp add: CYK-def subword-self)
lemma CYK-eq1 :
  \( i + 1 \leq \text{length } w \implies \text{CYK } G w i 1 = \{S. (S, \text{Leaf} (w!i)) \in \text{set } G\} \)
by (simp add: CYK-def, subst subword-nth2[simplified], assumption, subst Lang-eq1, rule refl)

theorem CYK-eq2 :
assumes A: \( i + j \leq \text{length } w \)
  and B: \( 1 < j \)
shows CYK G w i j = \{X | X A B k. (X, \text{Branch } A B) \in \text{set } G \land A \in \text{CYK } G w i k \land B \in \text{CYK } G w (i + k) \land \leq k \land k < j\}
proof (rule set-eqI, rule iffI, simp-all add: CYK-def)
fix X
assume a: \text{subword } w i j \in \text{Lang } G X
show \( \exists A B. (X, \text{Branch } A B) \in \text{set } G \land (\exists k. \text{subword } w i k \in \text{Lang } G A \land \text{subword } w (i + k) (j - k) \in \text{Lang } G B \land \text{Suc } 0 \leq k \land k < j) \)
proof
  have b: \( 1 < \text{length } (\text{subword } w i j) \) by (subst subword-length, rule A, rule B)
  note CYK-eq2[THEN iffD1, OF conj1, OF a b]
  then obtain A B l r where c: \( (X, \text{Branch } A B) \in \text{set } G \land \text{subword } w i j = l \cdot r \land l \in \text{Lang } G A \land r \in \text{Lang } G B \) by blast
  note Lang-no-Nil[OF c[THEN conjunct2, THEN conjunct2, THEN conjunct1]]
  hence d: \( 0 < \text{length } l \) by (case-tac l, simp-all)
  note Lang-no-Nil[OF c[THEN conjunct2, THEN conjunct2, THEN conjunct1]]
  hence e: \( 0 < \text{length } r \) by (case-tac r, simp-all)
  note subword-split2[OF c[THEN conjunct2, THEN conjunct1], OF A, OF d, OF e]
  with c show \( \exists \) thesis
proof (rule-tac \( x = A \) in exI, rule-tac \( x = B \) in exI, simp,
  rule-tac \( x = \text{length } l \) in exI, simp)
show \( \text{Suc } 0 \leq \text{length } l \land \text{length } l < j \) (is \( ?A \land ?B \))
proof
from d show \( ?A \) by (case-tac l, simp-all)
next
note arg-cong[where \( f = \text{length} \), OF c[THEN conjunct2, THEN conjunct1], THEN sym]
also have \( \text{length } (\text{subword } w i j) = j \) by (rule subword-length, rule A)
finally have \( \text{length } l + \text{length } r = j \) by simp
with e show \( ?B \) by force
qed
qed
qed
next
fix X
assume A B: \( (X, \text{Branch } A B) \in \text{set } G \land (\exists k. \text{subword } w i k \in \text{Lang } G A \land \text{subword } w (i + k) (j - k) \in \text{Lang } G B \land \text{Suc } 0 \leq k \land k < j) \)
then obtain A B k where a: \( (X, \text{Branch } A B) \in \text{set } G \land \text{subword } w i k \in \text{Lang } G A \land \text{subword } w (i + k) (j - k) \in \text{Lang } G B \land \text{Suc } 0 \leq k \land k < j \) by blast
show \text{subword } w \ i \ j \in \text{Lang G X}

\text{proof}\ (\text{rule Lang-eq2[THEN iffD2, THEN conjunct1]}, \text{rule-tac } x=\text{A in exI}, \text{rule-tac } x=\text{B in exI}, \text{simp add: } \text{a},
\quad \text{rule-tac } x=\text{subword } w \ i \ k \text{ in exI}, \text{rule-tac } x=\text{subword } w \ (i + k) \ (j - k) \text{ in exI}, \text{simp add: } \text{a},
\quad \text{rule subword-split, rule A})

\text{from } \text{a show } 0 < k \text{ by force}

\text{next}

\text{from } \text{a show } k < j \text{ by simp}

\text{qed}

\text{qed}

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 \text{mem} :: \text{'a \Rightarrow 'a list \Rightarrow bool}
where
\text{mem } \text{a } [] = \text{False} \ |
\text{mem } \text{a } (x \#xs) = (x = \text{a } \lor \text{mem } \text{a } \text{xs})

lemma \text{mem[simp]} :
\text{mem } \text{x } \text{xs} = (x \in \text{set } \text{xs})
by(induct-tac \text{xs}, \text{simp, force})

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

lemma \text{match-prods} :
(\text{X} \in \text{set} (\text{match-prods } \text{G } \text{ls } \text{rs})) =
\quad (\exists \text{A } \in \text{set } \text{ls}. \exists \text{B } \in \text{set } \text{rs}. (\text{X}, \text{Branch } \text{A B}) \in \text{set } \text{G})
by(induct-tac \text{G}, \text{clarsimp}, \text{rename-tac } \text{l } \text{r } \text{ps}, \text{case-tac } \text{r}, \text{force+})
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.

\[
\text{function inner :: ('n, 't) CNG ⇒ (nat × nat ⇒ 'n list) ⇒ nat ⇒ nat ⇒ nat ⇒ 'n list}
\]

\[
\text{where inner G T i k j =}
\]

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

\[
\text{by pat-completeness auto}
\]

\[
\text{termination}
\]

\[
\text{by(relation measure(λ(a, b, c, d, e). e − d), rule wf-measure, simp)}
\]

\[
\text{declare inner.simps[simp del]}
\]

\[
\text{lemma inner :}
\]

\[
(X ∈ set(inner G T i k j)) =
\]

\[
(∃l. k ≤ l ∧ l < j ∧ X ∈ set(match-prods G (T(i, l)) (T(i + l, j − l))))
\]

\[
\text{(is } (?L G T i k j) = (?R G T i k j))
\]

\[
\text{proof(induct-tac G T i k j rule: inner.induct)}
\]

\[
\text{fix G T i k j}
\]

\[
\text{assume a: } k < j \implies (?L G T i (k + 1) j) = (?R G T i (k + 1) j)
\]

\[
\text{show ?thesis}
\]

\[
\text{proof(case-tac k < j)}
\]

\[
\text{assume b: } k < j
\]

\[
\text{with a have c: } (?L G T i (k + 1) j) = (?R G T i (k + 1) j) \text{ by simp}
\]

\[
\text{show ?thesis}
\]

\[
\text{proof(subst inner.simps, simp add: b, rule iffI, erule disjE', rule-tac x=k in exI, simp add: b)}
\]

\[
\text{assume X ∈ set(inner G T i (Suc k) j)}
\]

\[
\text{with c have } (?R G T i (k + 1) j) \text{ by simp}
\]

\[
\text{thus } (?R G T i k j) \text{ by(clarsimp, rule-tac x=1 in exI, simp)}
\]

\[
\text{next}
\]

\[
\text{assume } (?R G T i k j)
\]

\[
\text{then obtain l where d: } k ≤ l ∧ l < j ∧ X ∈ set(match-prods G (T(i, l)) (T(i + l, j − l))) \text{ by blast}
\]

\[
\text{show X ∈ set(match-prods G (T(i, k)) (T(i + k, j − k))) ∨ (?L G T i (Suc k) j)
\]

\[
\text{proof(case-tac Suc k ≤ l, rule disjI2, subst c[simplified], rule-tac x=1 in exI, simp add: d,}
\]

\[
\text{rule disjI1)}
\]

\[
\text{assume ¬ Suc k ≤ l}
\]

\[
\text{with d have l = k by force}
\]

\[
\text{with d show X ∈ set(match-prods G (T(i, k)) (T(i + k, j − k))) by simp}
\]

\[
\text{qed}
\]

\[
\text{qed}
\]

16
next
assume $\neg k < j$
thus $\exists$thesis by (subst inner.simps, simp)
qed
qed

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
  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 $T'$)
by pat-completeness auto
termination
by (relation inv-image (less-than <*lex*> less-than) $(\lambda(a, b, c, d, e). (c - e, c - (d + e)))$, rule wf-inv-image, rule wf-lex-prod, (rule wf-less-than)+, simp-all)

declare main.simps[resp del]

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

proof –
  have $\forall \ length \ T' \ w. \ main \ G \ T \ len \ i \ j = T' \Longrightarrow \ length \ w = len \Longrightarrow 1 < j \Longrightarrow i + j \leq\ len \Longrightarrow$
    $(\forall j' < i, \forall i'. 1 \leq j' \Longrightarrow i' + j' \leq\ len \Longrightarrow set(T(i', j')) = CYK \ G \ w \ i' \ j')$
  →
    $(\forall i' < i, i' + j \leq\ len \Longrightarrow set(T(i', j)) = CYK \ G \ w \ i' \ j) \rightarrow$
    $(\forall j' \geq 1, \forall i'. i' + j' \leq\ len \Longrightarrow set(T'(i', j')) = CYK \ G \ w \ i' \ j') \ (\text{is} \ \forall \ length. \ ?P \ G \ T \ len \ i \ j)$
  proof (rule allI, induct-tac G T len i j rule: main.induct, (drule meta-spec, drule meta-mp, rule refl)+, clarify)
    fix $G$ $T$ $i$ $j$ $i'$ $j'$
    fix $w :: \ \text{a list}$
    assume $a: i + j <\ length \ w \Longrightarrow \ ?P \ G \ (T((i, j) := \text{inner } G \ T \ i \ 1 \ j)) \ (\length \ w) \ (i + 1) \ j$
assume \( b: \neg i + j < \text{length } w \implies j < \text{length } w \implies ?P G \ (T((i, j) := \text{inner } G T i j)) \) (length \( w \)) \( 0 \) (\( j + 1 \))

assume \( c: I < j \)

assume \( d: i + j \leq \text{length } w \)

assume \( e: (I :: \text{nat}) \leq j' \)

assume \( f: i' + j' \leq \text{length } w \)

assume \( g: \forall j' < j. \forall i'. I \leq j' \implies i' + j' \leq \text{length } w \implies \text{set}(T(i', j')) = CYK G w i' j' \)

assume \( h: \forall i' < i. i' + j \leq \text{length } w \implies \text{set}(T(i', j)) = CYK G w i' j \)

have \( \text{inner: set } (\text{inner } G T i (\text{Suc } 0) j) = CYK G w i j \)

proof\( (\text{rule set-eq1, subst inner, subst match-prods, subst CYK-eq2, rule } d, \text{ rule } c, \text{ simp}) \)

fix \( X \)

show \( \exists l \geq \text{Suc } 0. l < j \land (\exists A \in \text{set}(T(i, l), \exists B \in \text{set}(T(i + 1, j - l)) \land X, \land \text{Branch } A B) \in \text{set } G) \)

(\( \exists A B, (X, \text{Branch } A B) \in \text{set } G \land (\exists k. A \in CYK G w i k \land B \in CYK G w (i + k) (j - k) \land \text{Suc } 0 \leq k \land k < j) \)) \( (\text{is } ?L = ?R) \)

proof

assume \( ?L \)

do \( \text{thus } ?R \)

proof\( (\text{clarsimp, rule-tac } x = A \text{ in } \text{exI}, \text{rule-tac } x = B \text{ in } \text{exI}, \text{simp, rule-tac } x = l \)

in \( \text{exI}, \text{simp} \)

fix \( I A B \)

assume \( i: \text{Suc } 0 \leq l \)

assume \( j: l < j \)

assume \( k: A \in \text{set}(T(i, l)) \)

assume \( l: B \in \text{set}(T(i + l, j - l)) \)

note \( g[\text{rule-format, where } i' = i \land j' = ] \)

with \( \text{d i j} \) have \( A: \text{set}(T(i, l)) = CYK G w i l \) by force

note \( g[\text{rule-format, where } i' = i + l \land j' = j - l] \)

with \( \text{d i j} \) have \( \text{set}(T(i + l, j - l)) = CYK G w (i + l) (j - l) \) by force

with \( k l A \) show \( A \in CYK G w i l \land B \in CYK G w (i + l) (j - l) \) by simp

qed

next

assume \( ?R \)

do \( \text{thus } ?L \)

proof\( (\text{clarsimp, rule-tac } x = k \text{ in } \text{exI}, \text{simp}) \)

fix \( A B k \)

assume \( i: \text{Suc } 0 \leq k \)

assume \( j: k < j \)

assume \( k: A \in CYK G w i k \)

assume \( l: B \in CYK G w (i + k) (j - k) \)

assume \( m: (X, \text{Branch } A B) \in \text{set } G \)

note \( g[\text{rule-format, where } i' = i \land j' = ] \)

with \( \text{d i j} \) have \( A: CYK G w i k = \text{set}(T(i, k)) \) by force

note \( g[\text{rule-format, where } i' = i + k \land j' = j - k] \)

with \( \text{d i j} \) have \( CYK G w (i + k) (j - k) = \text{set}(T(i + k, j - k)) \) by force

with \( k l A \) have \( A \in \text{set}(T(i, k)) \land B \in \text{set}(T(i + k, j - k)) \) by simp
with \( m \) show \( \exists A \in \text{set}(T(i, k)). \exists B \in \text{set}(T(i + k, j - k)). (X, \text{Branch} A \ B) \in \text{set} G \) by force

c qed
c qed

c qed

c show set((main G T (length w) i j)(i', j')) = CYK G w i' j'

c proof\(\text{case-tac } i + j = \text{length } w\)

c assume \( i: i + j = \text{length } w \)

c show \( \text{thesis} \)

c proof\(\text{case-tac } j < \text{length } w\)

c assume \( j: j < \text{length } w \)

c show \( \text{thesis} \)

c proof\(\text{subst main..simps, simp add: Let-def i j,}

\rule b[rule-format, where w=w and i'=i' and j'=j', OF - refl, simplified],

\text{simp-all add: inner})

c from i show \( \neg i + j < \text{length } w \) by simp

next

c from c show \( 0 < j \) by simp

next

c from j show \( \text{Suc } j \leq \text{length } w \) by simp

next

c from e show \( \text{Suc } 0 \leq j' \) by simp

next

c from f show \( i' + j' \leq \text{length } w \) by assumption

next

fix \( i'' j'' \)

c assume \( k: j'' < \text{Suc } j \)

c assume \( l: \text{Suc } 0 \leq j'' \)

c assume \( m: i'' + j'' \leq \text{length } w \)

c show \( (i'' = i \longrightarrow j'' \neq j) \longrightarrow \text{set}(T(i'', j'')) = CYK G w i'' j'' \)

c proof\(\text{case-tac } j'' = j, \text{simp-all, clarify})

\ assume \( n: j'' = j \)

\ assume \( i'' \neq i \)

\ with \( i \ m \ n \) have \( i'' < i \) by simp

\ with \( n \ m \ h \) show \( \text{set}(T(i'', j)) = CYK G w i'' j \) by simp

next

\ assume \( j'' \neq j \)

\ with \( k \) have \( j'' < j \) by simp

\ with \( l \ m \ g \) show \( \text{set}(T(i'', j'')) = CYK G w i'' j'' \) by simp

c qed

c qed

next

\ assume \( \neg j < \text{length } w \)

\ with \( i \) have \( j: i = 0 \land j = \text{length } w \) by simp

\ show \( \text{thesis} \)

\ proof\(\text{subst main..simps, simp add: Let-def j, intro conjI, clarify})

\ from \( j \) and \( \text{inner} \) show \( \text{set } (\text{inner } G T 0 (\text{Suc } 0) (\text{length } w)) = CYK G w 0 \)
(length w) by simp

next
show 0 < i' --- set(T(i', j')) = CYK G w i' j'
proof
assume 0 < i'
with j and f have j' < j by simp
with e g f show set(T(i', j')) = CYK G w i' j' by simp
qed
next
show j' ≠ length w --- set(T(i', j')) = CYK G w i' j'
proof
assume j' ≠ length w
with j and f have j' < j by simp
with e g f show set(T(i', j')) = CYK G w i' j' by simp
qed
next
assume i + j ≠ length w
with d have i: i + j < length w by simp
show ?thesis
proof(subst main.simps, simp add: Let-def i,
rule a[rule-format, where w=w and i'=i' and j'=j', OF i, OF refl,
simplified])
  from c show Suc 0 < j by simp
next
  from i show Suc(i + j) ≤ length w by simp
next
    from c show Suc 0 ≤ j' by simp
next
    from f show i' + j' ≤ length w by assumption
next
fix i'' j''
assume j'' < j
and Suc 0 ≤ j''
and i'' + j'' < length w
with g show set(T(i'', j'')) = CYK G w i'' j'' by simp
next
fix i'' assume j: i'' < Suc i
show set(if i'' = i then inner G T i (Suc 0) j else T(i'', j)) = CYK G w i'' j
proof(simp split: if-split, rule conjI, clarify, rule inner, clarify)
assume i'' ≠ i
with j have i'' < i by simp
with d h show set(T(i'', j)) = CYK G w i'' j by simp
qed
qed
qed
with assms show ?thesis by force
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 ∈ set(init-match G a)) =
((X, Leaf a) ∈ set G)
by (induct-tac G a rule: init-match.induct, simp-all)

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 ⇒ ‘n list
where
  init' G [] k = emptyT |
  init' G (t#ts) k = (init' G ts (k + 1))(Suc k, 1) := init-match G t

lemma init' :
assumes i + 1 ≤ length w
shows set(init' G w 0 (i, 1)) = CYK G w i 1
proof −
  have ∀ i. Suc i ≤ length w →
    (∀ k. set(init' G w k (k + i, Suc 0))) = CYK G w i (Suc 0)
    (is ∀ i. ?P i w → (∀ k. ?Q i k w))
proof (induct-tac w, clarsimp+, rule conjI, clarsimp, rule set-eqI, subst init-match)
fix x w S
  show ((S, Leaf x) ∈ set G) = (S ∈ CYK G (x#w) 0 (Suc 0)) by (subst CYK-eq1[simplified], simp-all)
next
fix x w i
  assume a: ∀ i. ?P i w → (∀ k. ?Q i k w)
  assume b: i ≤ length w
  show 0 < i → (∀ k. set(init' G w (Suc k) (k + i, Suc 0))) = CYK G (x#w) i (Suc 0)
proof (clarify, case-tac i, simp-all, subst CYK-eq1[simplified], simp, erule subst, rule b, simp)
  fix k
assume $c : i = \text{Suc} \ j$

note $a$[rule-format, where $i = j$ and $k = \text{Suc} \ k$]
with $b$ and $c$ have $\text{set}(\text{init'} \ G \ w \ (\text{Suc} \ k) \ (\text{Suc} \ k + j, \ \text{Suc} \ 0)) = \text{CYK} \ G \ w \ j$ $(\text{Suc} \ 0)$ by simp
also with $b$ and $c$ have $\ldots = \{S. \ (S, \ \text{Leaf} \ (w ! j)) \in \text{set} \ G\}$ by (subst CYK-eq1[simplified], simp-all)
finally show $\text{set}(\text{init'} \ G \ w \ (\text{Suc} \ k) \ (\text{Suc} \ (k + j), \ \text{Suc} \ 0)) = \{S. \ (S, \ \text{Leaf} \ (w ! j)) \in \text{set} \ G\}$ by simp
qed

with assms have $\forall \ k. \ ?Q \ i \ k \ w$ by simp
note this[rule-format, where $k = 0$]
thus $\?\text{thesis}$ by simp
qed

The next version of initialization refines $\text{init'}$ 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, $\text{init-match}$ is an empty list). No initial table is then needed as such words can immediately be rejected.

fun $\text{init} :: (\text{'}n, \text{'t}) \text{CNG} \Rightarrow \text{'}t \text{ list} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat} \Rightarrow \text{'}n \text{ list}) \text{ option}$
where $\text{init} \ G \ [] \ k = \text{None}$ |
$\text{init} \ G \ [t] \ k = (\text{case} \ (\text{init-match} \ G \ t) \ of$ |
$\text{[]} \Rightarrow \text{None}$ |
$xs \Rightarrow \text{Some}(\text{emptyT}((k, \text{1}) := xs)))$ |
$\text{init} \ G \ (t \#ts) \ k = (\text{case} \ (\text{init-match} \ G \ t) \ of$ |
$\text{[]} \Rightarrow \text{None}$ |
$xs \Rightarrow (\text{case} \ (\text{init} \ G \ ts \ (k + \text{1})) \ of$ |
$\text{None} \Rightarrow \text{None}$ |
$\text{Some} \ T \Rightarrow \text{Some}(T((k, \text{1}) := xs)))$

lemma $\text{init1}$[rule-format] :
$\forall \ T. \ \text{init} \ G \ w \ k = \text{Some} \ T \rightarrow$
$\text{init'} \ G \ w \ k = T$
by(induct-tac $G \ w \ k$ rule: $\text{init}. \text{induct}, \ \text{clarsimp}+, \ \text{simp split: list}. \text{split-asm}, \ \text{rule ext}, \ \text{clarsimp},$ |
$\text{simp split: list}. \text{split-asm option.split-asm}, \ \text{rule ext}, \ \text{clarsimp}, \ \text{force}$)

lemma $\text{init2}$ :
$(\text{init} \ G \ w \ k = \text{None}) =$
$(w = \text{[]} \ \vee (\exists a \in \text{set} \ w. \ \text{init-match} \ G \ a = \text{[]}))$
by(induct-tac $G \ w \ k$ rule: $\text{init}. \text{induct}, \ \text{simp}, \ \text{simp split: list}. \text{split},$
$\text{simp split: list}. \text{split option.split-asm}, \ \text{rule ext}, \ \text{clarsimp}, \ \text{force}$)

4.3 The overall procedure

definition $\text{cyk} \ G \ S \ w = (\text{case} \ \text{init} \ G \ w \ 0 \ of$
\begin{align*}
\text{None } \Rightarrow \text{ False} \\
| \text{Some } T \Rightarrow \text{let } len = \text{length } w \text{ in} \\
\quad \text{if } len = 1 \text{ then mem } S \ (T(0, 1)) \\
\quad \text{else let } T' = \text{main } G \ T \ len \ 0 \ 2 \text{ in} \\
\quad \text{mem } S \ (T'(0, len))
\end{align*}

\texttt{theorem cyk : cyk} G S w = (w \in \text{Lang } G \ S) \\
\texttt{proof (simp add: cyk-def split: option.split, simp-all add: Let-def,} \\
\quad \texttt{rule conjI, subst init2, simp, rule conjI)} \\
\texttt{show } w = \textbf{[]} \rightarrow \textbf{[]} \notin \text{Lang } G \ S \texttt{by (clarify, drule Lang-no-Nil, clarify)} \\
\texttt{next} \\
\texttt{show } (\exists x \in \text{set } w. \ \text{init-match } G x = \textbf{[]}) \rightarrow w \notin \text{Lang } G \ S \texttt{by (clarify, drule Lang-term, subst (asm) init-match \ THEN \ sym, force)} \\
\texttt{next} \\
\texttt{show } \forall T. \ \text{init } G \ w \ 0 = \textbf{Some } T \rightarrow \\
\quad ((\text{length } w = \textbf{Suc } 0 \rightarrow S \in \text{set}(T(0, \textbf{Suc } 0))) \land \\
\quad (\text{length } w \neq \textbf{Suc } 0 \rightarrow S \in \text{set}(\text{main } G \ T \ (\text{length } w) \ 0 \ 2 \ (0, \text{length } w)))) = \\
\quad (w \in \text{Lang } G \ S) \ (\textbf{is } \forall T. \ ?P T \rightarrow \ ?L T = \ ?R) \\
\texttt{proof clarify} \\
\texttt{fix } T \\
\texttt{assume a: } ?P T \\
\texttt{hence b: init' } G \ w \ 0 = T \texttt{by (rule init1)} \\
\texttt{note init2[THEN iffD2, OF disjI1]} \\
\texttt{have c: } w \neq \textbf{[]} \texttt{by (clarify, drule init2[where G=G and k=0, THEN iffD2, OF disjI1], simp add: a)} \\
\texttt{have } ?L (\text{init' } G \ w \ 0) = \ ?R \\
\texttt{proof (case-tac length } w = 1, \ \text{simp-all)} \\
\texttt{assume d: } \text{length } w = \textbf{Suc } 0 \\
\texttt{show } S \in \text{set}(\text{init' } G \ w \ 0 \ (0, \textbf{Suc } 0)) = \ ?R \\
\texttt{by (subst init' [simplified], simp add: d, subst CYK-Lang \ THEN \ sym, simp add: d)} \\
\texttt{next} \\
\texttt{assume length } w \neq \textbf{Suc } 0 \\
\texttt{with c have } 1 < \text{length } w \texttt{by (case-tac } w, \ \text{simp-all)} \\
\texttt{hence d: } \text{Suc}(\textbf{Suc } 0) \leq \text{length } w \texttt{by simp} \\
\texttt{show } (S \in \text{set}(\text{main } G \ (\text{init' } G \ w \ 0) \ (\text{length } w) \ 0 \ 2 \ (0, \text{length } w))) = (w \in \text{Lang } G \ S) \\
\texttt{proof (subst main, simp-all, rule d)} \\
\texttt{fix } i'\ j' \\
\texttt{assume j' < 2 \ and } \textbf{Suc } 0 \leq j' \\
\texttt{hence c: } j' = 1 \texttt{ by simp} \\
\texttt{assume i' + j' \leq } \text{length } w \\
\texttt{with e have f: } i' + 1 \leq \text{length } w \texttt{ by simp} \\
\texttt{have set(\text{init' } G \ w \ 0 \ (i', 1)) = CYK G w i' 1 \texttt{ by (rule init', rule f)} \\
\texttt{with e show set(\text{init' } G \ w \ 0 \ (i', j')) = CYK G w i' j' \texttt{ by simp}} \\
\texttt{next}
from d show Suc 0 ≤ length w by simp
next
  show (S ∈ CYK G w 0 (length w)) = (w ∈ Lang G S) by (rule CYK-Lang)
qed
qed
with b show ?L T = ?R by simp
qed
qed

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", "b", "a"],
  ["b", "a", "b", "a"]

end

References