A formalisation of the Cocke-Younger-Kasami algorithm

Maksym Bortin

December 17, 2016

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.

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


type-synonym ('n,'t) CNG = ('n × ('n,'t) RHS) list

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 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 ⇒ ('(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 ∧
(case rhs of
Branch A B ⇒ x = l · [Inl A, Inl B] · r
| Leaf t ⇒ x = l · [Inr t] · r)}

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

\[ \text{definition } \text{Lang} :: (n', t') \text{ CNG} \Rightarrow (n', t') \text{ word set} \]
\[ \text{where } \text{Lang } G = S = \{ w. [\text{Inl } S] \rightarrow G \} \]

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

\[ \text{definition } G-anbn = \]
\[ (["S", \text{Branch } "A" "B"],
(["S", \text{Branch } "A" "T"],
(["T", \text{Branch } "S" "B"],
(["A", \text{Leaf } "a"],
(["B", \text{Leaf } "b"])
\]

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

\[ \text{lemma prod-into-DSTEP1} : \]
\( (S, \text{Branch } A B) \in \text{set } G \Rightarrow \)
\( L \cdot [\text{Inl } S] \cdot R - G \rightarrow L \cdot [\text{Inl } A, \text{Inl } B] \cdot R \)
\[ \text{by (simp add: DSTEP-def, rule-tac } x=L \text{ in } exI, \text{ force) } \]

\[ \text{lemma prod-into-DSTEP2} : \]
\( (S, \text{Leaf } a) \in \text{set } G \Rightarrow \)
\( L \cdot [\text{Inl } S] \cdot R - G \rightarrow L \cdot [\text{Inr } a] \cdot R \)
by(simp add: DSTEP-def, rule-tac x=L in exI, force)

lemma DSTEP-D:
s −G→ t ⟹
exists L N R rhs. s = L · [Inl N] · R ∧ (N, rhs) ∈ set G ∧
(∀ A B. rhs = Branch A B ⟹ t = L · [Inl A, Inl B] · R) ∧
(∀ x. rhs = Leaf x ⟹ t = L · [Inr x] · R)
by(unfold DSTEP-def, clarsimp, simp split: RHS.split-asm, blast+)

lemma DSTEP-append:
assumes a: s −G→ t
shows L · s · R −G→ L · t · R
proof –
from a have ∃ l N r rhs. s = l · [Inl N] · r ∧ (N, rhs) ∈ set G ∧
(∀ A B. rhs = Branch A B ⟹ t = l · [Inl A, Inl B] · r) ∧
(∀ x. rhs = Leaf x ⟹ t = l · [Inr x] · r) (is ∃ l N r rhs. ?P l N r rhs)
by(rule DSTEP-D)
then obtain l N r rhs where ?P l N r rhs by blast
thus ?thesis by(simp add: DSTEP-def, rule-tac x=L in exI,
rule-tac x=N in exI, rule-tac x=r · R in exI,
simp, rule-tac x=rhs in exI, simp split: RHS.split)
qed

lemma DSTEP-star-mono:
s −G→* t ⟹ length s ≤ length t
proof(erule rtrancl_induct, simp)
fix t u
assume s −G→* t
assume a: t −G→ u
assume b: length s ≤ length t
show length s ≤ length u
proof –
from a have ∃ L N R rhs. t = L · [Inl N] · R ∧ (N, rhs) ∈ set G ∧
(∀ A B. rhs = Branch A B ⟹ u = L · [Inl A, Inl B] · R) ∧
(∀ x. rhs = Leaf x ⟹ u = L · [Inr x] · R) (is ∃ L N R rhs.
?P L N R rhs)
by(rule DSTEP-D)
then obtain L N R rhs where ?P L N R rhs by blast
with b show ?thesis by(case-tac rhs,clarsimp+)
qed
lemma DSTEP-comp :
assumes a : l ⋅ r −→ G → t
shows ∃ l’ r’. l −→ G I= l’ ∧ r −→ G I= r’ ∧ t = l’ ⋅ r’
proof −
from a have ∃ L N R rhs. l ⋅ r = L ⋅ [Inl N] ⋅ R ∧ (N, rhs) ∈ set G ∧
(∀ A B. rhs = Branch A B −→ t = L ⋅ [Inl A, Inl B] ⋅ R) ∧
(∀ x. rhs = Leaf x −→ t = L ⋅ [Inr x] ⋅ R) (is ∃ L N R rhs. ?T L N R rhs)
by (rule DSTEP-D)
then obtain L N R rhs where b: ?T L N R rhs by blast
hence l ⋅ r = L · Inl N # R by simp
by (rule append-eq-append-conv2[THEN iffD1])
then obtain xs where c: l = L · xs ∧ xs · r = Inl N # R ∨ l · xs = L ∧ r =
xs · Inl N # R (is ?C1 ∨ ?C2) by blast
show ?thesis
proof (cases rhs)
case (Leaf x)
with b have d: t = L ⋅ [Inr x] ⋅ R ∧ (N, Leaf x) ∈ 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: R.HS.split, blast)
next
case (Cons z zs) with d and e show ?thesis
by (erule_tac x=L in exI, clarsimp, simp add: DSTEP-def, simp split: R.HS.split, blast)
qed
next
assume e: ?C2
show ?thesis
proof (cases xs)
case Nil with d and e show ?thesis
by (erule_tac x=L in exI, clarsimp, simp add: DSTEP-def, simp split: R.HS.split, blast)
next
case (Cons z zs) with d and e show ?thesis
by (erule_tac x=L in exI, clarsimp, simp add: DSTEP-def, simp split: R.HS.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=Leaf x in 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 e show ?thesis
proof
  assume c: ?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 in exI, clarsimp, simp add: DSTEP-def, simp split: RHS.split, blast)
  qed
next
  assume c: ?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 add: DSTEP-def, simp split: RHS.split, blast)
  qed
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 l·s · r −G→∗ t
  proof
  have ∃ r. l · r = (l' · r') (is l·s)
  proof (erule rtrancl_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 ∙ r
  with b have ∃ l’ r’. l’ ∙ r’ ∨ s ∙ r’ ∧ t = l’ ∙ r’ by simp
  then obtain l’ r’ where c: l’ ∙ r’ ∨ s ∙ r’ ∧ t = l’ ∙ r’ by blast
  with a have l’ ∙ r’ − G → u by simp
  hence ∃ l” r”. l” ∨ r” − G → u = l” ∙ r” by (rule DSTEP-comp)
  then obtain l” r” where l” ∨ r” − G → u = l” ∙ r” by blast
  with c show ∃ l’ r’. l’ ∨ r” − G → u = l’ ∙ r” by blast
  by (rule-tac x=l’ in exI, rule-tac x=r’ in exI, force)
qed

next
  fix s t
  assume a: s − G → t
  assume b: ?Q l s
  show ?Q l t
  proof (clarsimp)
    fix r r’
    assume r − G → r’ with b have c: l ∨ r − G → s ∨ r’ by simp
    with a have c: l ∨ r − G → l ∨ r’ by simp
    with c show ?thesis by simp
    qed
    qed
  qed

theorem DSTEP-star-comp2 : 
assumes A: l − G →* l’
  and B: r − G →* r’
shows l ∙ r − G →* l’ ∙ r’
proof
  have l − G →* l’ l’ ∙ r’ −→∀ r r’ r − G → r’ −→ l ∙ r − G →* l’ ∙ r’ (is ?P l l’ −→ ?Q l l’)
proof (erule rtrancl-induct)
  show ?Q l l
proof (clarify, erule rtrancl-induct, simp)
    fix r s t
    assume a: s − G → t
    assume b: l ∙ r − G →* l ∙ s
    show l ∙ r − G →* l ∙ t
    proof
      from a have l ∙ s − G → l ∙ t by (drule-tac L=l and R=[] in DSTEP-append, simp)
      with b show ?thesis by simp
      qed
      qed
    qed
next
  fix s t
  assume a: s − G → t
  assume b: ?Q l s
  show ?Q l t
  proof (clarsimp)
    fix r r’
    assume r − G →* r’ with b have c: l ∨ r − G → s ∨ r’ by simp

from a have s · r' −→ t · r′ by \( \text{drule-tac \big\{L=[] and R=r' in DSTEP-append, simp\} \)
with c show l · r −→* t · r′ by simp
qed
qed
with A and B show ?thesis by simp
qed

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

8
2.1 Properties of generated languages

**lemma** Lang-no-Nil:

\[ w \in \text{Lang } G S \implies w \neq [] \]

by (simp add: Lang-def, drule trancl-into-rtrancl, drule DSTEP-star mono, force)

**lemma** Lang-rtrancl-eq:

\[ (w \in \text{Lang } G S) = \text{map Inr } w \]

proof (simp add: Lang-def, rule iffI, erule trancl-into-rtrancl)

assume \( ?p \in ?R^\sim \)

hence \( ?p \in (?R^\sim)^\sim \) by (subs rtrancl-trancl-refcl[THEN sym], assumption)

hence \( [\text{Inl } S] = \text{map Inr } w \) by force

thus \( ?p \in ?R^\sim \) by (case-tac w, simp-all)

qed

**lemma** Lang-term:

\[ w \in \text{Lang } G S \implies \forall x \in \text{set } w. \exists N. (N, \text{Leaf } x) \in \text{set } G \]

by (clarsimp simp add: Lang-def, drule DSTEP-trancl-term, simp, erule imageI, assumption)

**lemma** Lang-eq1:

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

proof (simp add: Lang-def, rule iffI, subst (asm) trancl-unfold-left, clapsimp)

fix t

assume a: \([\text{Inl } S] \rightarrow t \)

assume b: \( t \rightarrow S \rightarrow \text{map Inr } x \)

note DSTEP-star mono[OF b, simplified]

hence c: \text{length } t \leq 1 \text{ by simp}

have \( \exists z. t = [z] \)

proof (cases t)

assume t = []

with b have d: \( [] \rightarrow S \rightarrow \text{map Inr } x \) by simp

have \( \forall s. ([], s) \in (\text{DSTEP } G)^* \implies s = [] \)

by (erule rtrancl-induct, simp-all, drule DSTEP-D, clapsimp)

note this[OF d]

thus \( ?p \) thesis by simp

next

fix z zs

assume t = z#zs
with \( c \) show \(?thesis\) by force

qed

with \( a \) have \( \exists z. \ (S, \text{Leaf } z) \in \text{set } G \wedge t = [\text{Inr } z] \)
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 \( [\text{Inr } z] \to ^* [\text{Inr } x] \)

hence \( ([\text{Inr } z], [\text{Inr } x]) \in (\text{(DSTEP } G)^+) \) by simp
hence \( [\text{Inr } z] = [\text{Inr } x] \lor [\text{Inr } z] \to [\text{Inr } x] \) by force
hence \( x = z \)

proof
assume \( [\text{Inr } z] = [\text{Inr } x] \) thus \(?thesis\) by simp
next
assume \( [\text{Inr } z] \to [\text{Inr } x] \)

hence \( \exists u. \ [\text{Inr } z] \to u \land u \to [\text{Inr } x] \) by(subst (asm) trancl-unfold-left, force)
then obtain \( u \) where \( [\text{Inr } z] \to u \) 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 \( [\text{Inl } S] \to ^* [\text{Inr } x] \)
by(rule r-into-trancl, simp add: DSTEP-def, rule-tac x=\[] in exI, rule-tac x=S in exI, rule-tac x=\[] in exI, simp, rule-tac x=Leaf x in exI, 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 \( \not= L \ not R \))
proof(rule iffI, clarify, subst (asm) Lang-def, simp, subst (asm) trancl-unfold-left, clarsimp)
have map-Inr-split : \( \bigwedge xs. \forall zs w. \text{map Inr } w = w :: zs \to\)
\((\exists u v. \ w = u \cdot v \land w = \text{map Inr } u \land zs = \text{map Inr } v)\)
by(induct-tac xs, simp, force)
fix \( t \)
assume \( a: \ \text{Suc } 0 < \text{length } w \)
assume \( b: \ [\text{Inl } S] \to ^* t \)
assume \( c: \ t \to ^* \text{map Inr } w \)
from \( b \) have \( \exists A B. \ (S, \text{Branch } A \ B) \in \text{set } G \land t = [\text{Inl } A, \text{Inl } B] \)
proof (simp add: DSTEP-def, clarify, case-tac l, simp-all, simp split: RHS.split-asn, clarify)
  fix x
  assume t = [Inr x]
  with c have d: [Inr x] \rightarrow^* map Inr w by simp
  have \( \forall x \cdot [Inr x] \rightarrow^* s \implies s = [Inr x] \)
  by (erule rtrancl-induct, simp-all, drule DSTEP-D, clarsimp, case-tac L, simp-all)
  note this[OF d]
  hence \( w = [x] \) by (case-tac w, simp-all)
  with a show False by simp
qed

then obtain A B where d: (S, Branch A B) \in set G \land t = [Inl A, Inl B] by blast
with c have e: [Inl A] \cdot [Inl B] \rightarrow^* map Inr w by simp
note DSTEP-star-compI[OF e]
then obtain \( l' r' \) where e: [Inl A] \rightarrow^* l' \land [Inl B] \rightarrow^* r' \land
  \( \forall x \cdot [Inr x] \rightarrow^* \) by blast
note map-Inr-split[rule-format, OF e] \then conjunct2, THEN conjunct2]
then obtain u v where f: \( w = u \cdot v \land l' = map Inr u \land r' = map Inr v \) by blast
with e have g: [Inl A] \rightarrow^* map Inr u \land [Inl B] \rightarrow^* map Inr v by simp
show \(?R
by(rule-tac x=A in exI, rule-tac x=B in exI, simp add: d,
  rule-tac x=u in exI, rule-tac x=v in exI, simp add: f,
  (subst Lang-rtrancl-eq)+, rule g)

next
assume \(?R
then obtain A B l r where a: (S, Branch A B) \in set G \land w = l \cdot r \land l \in Lang G A \land r \in Lang G B by blast
have [Inl A] \cdot [Inl B] \rightarrow^* map Inr l \cdot map Inr r
  by(rule DSTEP-star-comp2, subst Lang-rtrancl-eq[THEN sym], simp add: a,
    subst Lang-rtrancl-eq[THEN sym], simp add: a)
hence b: [Inl A] \cdot [Inl B] \rightarrow^* map Inr w by(simp add: a)
have c: w \in Lang G S
  by(simp add: Lang-def, subst trancl-unfold-left, rule-tac b=[Inl A] \cdot [Inl B] in relcompI,, simp add: DSTEP-def, rule-tac x=[] in exI, rule-tac x=S in exI, rule-tac x=[] in exI,
    simp, rule-tac x=Branch A B in exI, simp add: a[THEN conjunct1], rule b)
thus \(?L
proof
  show \( l < \) length w
  proof (simp add: a, rule ccontr, drule leI)
    assume length l + length r \leq Suc 0
    hence \( l = [] \lor r = [] \) by (case-tac l, simp-all)
    thus False
  proof
    assume l = []
    with a have [] \in Lang G A by simp
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 \( k=(0 :: \text{nat}) \), simplified]
  with \( a \) have \( x = w!i \) by simp
  with \( a \) show \( ?\text{thesis} \) by simp
qed
lemma subword-self:
subword w 0 (length w) = w
by(simp add: subword-def)

lemma take-split[rule-format]:
\forall n m. n \leq length xs \rightarrow n \leq m \rightarrow
take n xs \cdot 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 \leq length w \implies 0 < k \implies k < j \implies
subword w i j = subword w i k \cdot subword w (i + k) (j - k)
by(simp add: subword-def, subst take-split[where n=k, THEN sym], simp-all,
rule-tac f=\lambda x. take (j - k) (drop x w) in arg-cong, simp)

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) \land 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 \leq length w by force
  have subword w i j = subword w i (length l) \cdot subword w (i + length l) (j - length l)
    by(rule subword-split, rule B, rule C, rule b)
  with A have d: l \cdot r = subword w i (length l) \cdot 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 \in CYK G w 0 (length w)) = (w \in Lang G S)
by(simp add: CYK-def subword-self)
lemma CYK-eq1:
\[ i + 1 \leq \text{length } w \implies CYK G \ w \ i \ j = \{ S, (S, \text{Leaf } (w!i)) \} \in \text{set } G \]
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 \mid X A B k. \ (X, \text{Branch } A B) \in \text{set } G \wedge A \in CYK G \ w \ i \ k \wedge B \in CYK G \ w \ (i + k) \ (j - k) \land 1 \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 \[ \exists A B. \ (X, \text{Branch } A B) \in \text{set } G \wedge (\exists k. \ \text{subword } w \ i \ k \in \text{Lang } G \ A \wedge \text{subword } w \ (i + k) \ (j - k) \in \text{Lang } G \ B \wedge \text{Suc } 0 \leq k \wedge k < j) \]
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: \( (X, \text{Branch } A B) \in \text{set } G \wedge \text{subword } w \ i \ j = l \cdot r \wedge l \in \text{Lang } G \ A \wedge r \in \text{Lang } G \ B \) 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 conjunct2]]
  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 \( \text{thesis} \)
  proof(rule-tac x=A in exI, rule-tac x=B in exI, simp,
           rule-tac x=\text{length } l \in \text{exI}, simp)
    show Suc 0 \leq \text{length } l \wedge \text{length } l < j \ (\text{is } A \wedge ?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
  next
  fix X
  assume \[ \exists A B. \ (X, \text{Branch } A B) \in \text{set } G \wedge (\exists k. \ \text{subword } w \ i \ k \in \text{Lang } G \ A \wedge \text{subword } w \ (i + k) \ (j - k) \in \text{Lang } G \ B \wedge \text{Suc } 0 \leq k \wedge k < j) \]
  then obtain A B k where a: \( (X, \text{Branch } A B) \in \text{set } G \wedge \text{subword } w \ i \ k \in \text{Lang } G \ A \wedge \text{subword } w \ (i + k) \ (j - k) \in \text{Lang } G \ B \wedge \text{Suc } 0 \leq k \wedge k < j \) by blast
show subword w i j ∈ Lang G X
proof (rule Lang-eq2, THEN iffD2, THEN conjunct1),
rule-tac x=A in exI, rule-tac x=B in exI,
simp add: a,
rule-tac x=subword w i k in exI,
rule-tac x=subword w (i + k) (j − k) in exI,
simp add: a,
rule subword-split, rule A)
from a show 0 < k by force
next
from a show k < j by simp
qed
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 mem :: 'a ⇒ 'a list ⇒ bool
where
mem a [] = False |
mem a (x#xs) = (x = a ∨ mem a xs)

lemma mem[simp] :
mem x xs = (x ∈ set xs)
by(induct-tac xs, 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 match-prods :: ('n, 't) CNG ⇒ 'n list ⇒ 'n list ⇒ 'n list
where
match-prods [] ls rs = [] |
match-prods ((X, Branch A B)#ps) ls rs =
(if mem A ls ∧ mem B rs then X # match-prods ps ls rs
else match-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)
by(induct-tac G, clarsimp+, rename-tac l r ps, case-tac r, 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.

\[
\text{function} \quad \text{inner} :: (\text{'n}, \text{'t}) \rightarrow \text{nat} \times \text{nat} \rightarrow \text{'n list} \rightarrow \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \rightarrow \text{'n list}
\]

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

by \text{pat-completeness auto}

termination

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

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

lemma \text{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 \(\text{?L} G T i k j = \text{?R} G T i k j\))

proof (induct-tac \(G T i k j\) rule: inner.induct)

fix \(G T i k j\)

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

show \(\text{?thesis}\)

proof (case-tac \(k < j\))

assume \(b: k < j\)

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

show \(\text{?thesis}\)

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

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

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

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

next

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

then obtain \(l\) where \(d: k \leq l \land l < j \land X \in \text{set}(\text{match-prods} G (T(i, l)) (T(i + l, j - l)))\) by blast

show \(X \in \text{set}(\text{match-prods} G (T(i, k)) (T(i + k, j - k))) \lor \text{?L} G T i (\text{Suc} k) j\)

proof (case-tac \text{Suc} \(k \leq l\), rule disjI2, subst \(c[\text{simplified}]\), rule-tac \(x=l\) \text{ in} \text{exI}, simp add: d, rule disjI1)

assume \(\neg \text{Suc} k \leq l\)

with \(d\) have \(l = k\) by force

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

qed

qed

16
next
\begin{itemize}
\item assume \( k < j \)
\item thus \( \exists \theta \) thesis by (subst inner.simps, simp)
\end{itemize}
qed

Now the main part of the algorithm just iterates through all subwords up to the given length \( \text{len} \), calls \( \text{inner} \) 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 \( \text{main} :: \ (\text{'n}, \text{'t}) \ CNG \Rightarrow (nat \times \text{nat} \Rightarrow \text{'n list}) \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow (nat \times \text{nat} \Rightarrow \text{'n list}) \) where \( \text{main} \ G \ T \ \text{len} \ i \ j \) (let \( T' = T((i, j) := \text{inner} \ G \ T \ i \ 1 \ j) \) in
\begin{itemize}
\item if \( i + j < \text{len} \) then \( \text{main} \ G \ T' \ \text{len} \ (i + 1) \ j \)
\item else if \( j < \text{len} \) then \( \text{main} \ G \ T' \ \text{len} \ 0 \ (j + 1) \)
\end{itemize}
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 \( \text{main}.\text{simps}[\text{simp del}] \)

lemma \( \text{main} : \)
\begin{itemize}
\item assumes \( 1 < j \)
\item and \( i + j \leq \text{length} \ w \)
\item and \( \bigwedge i', j'. j' < j \implies 1 \leq j' \implies i' + j' \leq \text{length} \ w \implies \text{set}(T(i', j')) = CYK \ G \ w \ i' \ j' \)
\end{itemize}
\begin{itemize}
\item and \( \bigwedge i', i' < i \implies i' + j \leq \text{length} \ w \implies \text{set}(T(i', j)) = CYK \ G \ w \ i' \ j \)
\item and \( 1 \leq j' \)
\item and \( i' + j' \leq \text{length} \ w \)
\end{itemize}
\begin{itemize}
\item shows \( \text{set}(\text{(main} \ G \ T' \ (\text{length} \ w) \ i \ j)(i', j')) = CYK \ G \ w \ i' \ j' \)
\end{itemize}
\begin{itemize}
\item proof –
\item have \( \forall \text{len} \ T' \ w. \ \text{main} \ G \ T \ \text{len} \ i \ j = T' \rightarrow \text{length} \ w = \text{len} \rightarrow 1 < j \rightarrow i + j \leq \text{len} \)
\item \( \rightarrow (\forall i', j. 1 \leq j' \implies i' + j' \leq \text{len} \rightarrow \text{set}(T(i', j')) = CYK \ G \ w \ i' \ j') \)
\item \( \rightarrow (\forall i'. i' + j \leq \text{len} \rightarrow \text{set}(T(i', j)) = CYK \ G \ w \ i' \ j) \rightarrow (\forall j'. 1 \leq i'. i' + j' \leq \text{len} \rightarrow \text{set}(T'(i', j')) = CYK \ G \ w \ i' \ j') \) (is \( \forall \text{len.} \ ?P \ G \ T \ \text{len} \ i \ j \))
\item proof (rule allI, induct-tac \( G \ T \ \text{len} \ i \ j \) rule: main.induct, (drule meta-spec, drule meta-mp, rule refl+), clarify)
\item fix \( G \ T \ i \ j \ i' \ j' \)
\item fix \( w :: \text{'a list} \)
\item assume \( a: i + j < \text{length} \ w \ \Rightarrow ?P \ G \ (T((i, j) := \text{inner} \ G \ T \ i \ 1 \ j)) \) (length \( w \) \( i + 1 \) \( j \))
\end{itemize}
assume \( h: \neg i + j < \text{length } w \implies j < \text{length } w \implies \exists P \ G \ (T((i, j) := \text{inner } G \ T i j)) \) (length w) 0 (j + 1)

assume \( c: 1 < j \)
assume \( d: i + j \leq \text{length } w \)
assume \( e: (1::\text{nat}) \leq j' \)
assume \( f: i' + j' \leq \text{length } w \)
assume \( g: \forall j' < j. \forall i'. I \leq j' \implies i' + j' \leq \text{length } w \implies \text{set}(T(i', j')) = \text{CYK } G w i' j' \)
assume \( h: \forall i < i', i' + j \leq \text{length } w \implies \text{set}(T(i', j)) = \text{CYK } G w i' j \)

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

proof(rule set-eql, subst inner, subst match-prods, subst \text{CYK}eq2, rule d, rule 
c, simp)
fix \( X \)
show \( \exists l \geq \text{Suc } 0. l < j \wedge (\exists A \in \text{set}(T(i, l)). \exists B \in \text{set}(T(i + l, j - l)). \ (X, \text{Branch } A \ B) \in \text{set } G)) \) =
(\( \exists A \ B. \ (X, \text{Branch } A \ B) \in \text{set } G \wedge (\exists k. A \in \text{CYK } G w i k \wedge B \in \text{CYK } G w (i + k) (j - k) \wedge \text{Suc } 0 \leq k \wedge k < j) \)) (is \( \ ?L = \ ?R \))
proof
assume \( \ ?L \)
thus \( \ ?R \)
proof(clarsimp, rule-tac \( x=A \ in \ exI, \) rule-tac \( x=B \ in \ exI, \) simp, rule-tac \( x=l \)
in \( exI, \) simp)
fix \( l \ A \ B \)
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 \( g[\text{rule-format}, \text{where } i'=i \text{ and } j'=l] \)
with \( d \ i \ j \) have \( A: \text{set}(T(i, l)) = \text{CYK } G w i l \) by force
note \( g[\text{rule-format}, \text{where } i'=i + l \text{ and } j'=j - l] \)
with \( d \ i \ j \) have \( \text{set}(T(i + l, j - l)) = \text{CYK } G w (i + l) (j - l) \) by force
with \( k \ l \ A \) show \( A \in \text{CYK } G w i l \wedge B \in \text{CYK } G w (i + l) (j - l) \) by simp
qed
next
assume \( \ ?R \)
thus \( \ ?L \)
proof(clarsimp, rule-tac \( x=k \ in \ exI, \) simp)
fix \( A \ B \ k \)
assume \( i: \text{Suc } 0 \leq k \)
assume \( j: k < j \)
assume \( k: A \in \text{CYK } G w i k \)
assume \( l: B \in \text{CYK } G w (i + k) (j - k) \)
assume \( m: (X, \text{Branch } A \ B) \in \text{set } G \)
note \( g[\text{rule-format}, \text{where } i'=i \text{ and } j'=k] \)
with \( d \ i \ j \) have \( A: \text{CYK } G w i k = \text{set}(T(i, k)) \) by force
note \( g[\text{rule-format}, \text{where } i'=i + k \text{ and } j'=j - k] \)
with \( d \ i \ j \) have \( \text{CYK } G w (i + k) (j - k) = \text{set}(T(i + k, j - k)) \) by force
with \( k \ l \ A \) have \( A \in \text{set}(T(i, k)) \wedge B \in \text{set}(T(i + k, j - k)) \) by simp

18
with m show ∃A ∈ set(T(i, k)). ∃B ∈ set(T(i + k, j − k)). (X, Branch A B) ∈ set G by force
qed
qed
qed

show set((main G T (length w) i j)(i', j')) = CYK G w i' j'
proof (case-tac i + j = length w)
assume i: i + j = length w
show ?thesis
proof (case-tac j < length w)
assume j: j < length w
show ?thesis
proof (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],
simp-all add: inner)
from i show ¬ i + j < length w by simp
next
from c show 0 < j by simp
next
from j show Suc j ≤ length w by simp
next
from e show Suc 0 ≤ j' by simp
next
from f show i' + j' ≤ length w by assumption
next
fix i'' j''
assume k: j'' < Suc j
assume l: Suc 0 ≤ j''
assume m: i'' + j'' ≤ length w
show (i'' = i → j'' ≠ j) → set(T(i'' j'')) = CYK G w i'' j''
proof (case-tac j'' = j, simp-all, clarify)
assume n: j'' = j
assume i'' ≠ i
with i m n have i'' < i by simp
with n m h show set(T(i'', j)) = CYK G w i'' j by simp
next
assume j'' ≠ j
with k have j'' < j by simp
with l m g show set(T(i'', j'')) = CYK G w i'' j'' by simp
qed
qed
next
assume ¬ j < length w
with i have j: i = 0 ∧ j = length w by simp
show ?thesis
proof (subst main..simps, simp add: Let-def j, intro conj1, clarify)
from j and inner show set (inner G T 0 (Suc 0) (length w)) = CYK G w 0

19
(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 k show set(T(i'', j)) = CYK G w i'' j by simp

qed

qed

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 k show set(T(i'', j)) = CYK G w i'' j by simp

qed

qed

qed

with assms show ?thesis by force

20
4.2 Initialisation phase

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

\textbf{fun init-match ::} (\texttt{'n, 't}) CNG \texttt{$\Rightarrow$ 't $\Rightarrow$ 'n list}

\textbf{where}

\begin{align*}
\text{init-match} & \; [[] \; t = [[]] \mid \\
& \quad \text{init-match} \; ((X, \text{Branch A B})\#ps) \; t = \text{init-match} \; ps \; t \mid \\
& \quad \text{init-match} \; ((X, \text{Leaf a})\#ps) \; t = (\text{if} \; a = t \; \text{then} \; X \; \text{# init-match} \; ps \; t \\
& \quad \quad \quad \text{else init-match} \; ps \; t)
\end{align*}

\textbf{lemma init-match :}

\begin{align*}
(X \in \text{set(init-match} \; G \; a)) &= \\
((X, \text{Leaf a}) \in \text{set} \; G)
\end{align*}

\textbf{by}(\text{induct-tac} \; G \; a \; \text{rule: init-match.induct, simp-all})

Representing the empty table.

\textbf{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.

\textbf{fun init' ::} (\texttt{'n, 't}) CNG \texttt{$\Rightarrow$ 't list $\Rightarrow$ nat $\Rightarrow$ nat $\Rightarrow$ 'n list}

\textbf{where}

\begin{align*}
\text{init'} \; G \; [[] \; k = \text{emptyT} \mid \\
& \quad \text{init'} \; G \; (t\#ts) \; k = (\text{init'} \; G \; ts \; (k + 1))((k, 1) := \text{init-match} \; G \; t)
\end{align*}

\textbf{lemma init' :}

\textbf{assumes} \; i + 1 \leq \text{length} \; w

\textbf{shows} set(\text{init'} \; G \; w \; 0 \; (i, 1)) = \text{CYK} \; G \; w \; i \; 1

\textbf{proof –}

\textbf{have} \; \forall \; i. \; \text{Suc} \; i \leq \text{length} \; w \; \longrightarrow \\
\quad \quad (\forall k. \; \text{set(} \text{init'} \; G \; w \; k \; (k + i, \; \text{Suc} \; 0)) = \text{CYK} \; G \; w \; i \; (\text{Suc} \; 0)) \; (\text{is} \forall \; i. \; ?P \; i \; w \\
\quad \quad \quad \quad \quad \quad \longrightarrow (\forall k. \; ?Q \; i \; k \; w))

\textbf{proof(induct-tac} \; w, \text{clarsimp+}, \text{rule conjI, clarsimp, rule set-eqI, subst init-match})

\textbf{fix} \; x \; w \; S

\textbf{show} \; ((S, \; \text{Leaf} \; x) \in \text{set} \; G) = (S \in \text{CYK} \; G \; (x\#w) \; 0 \; (\text{Suc} \; 0)) \; \text{by(subst \text{CYK-egI[simplified], simp-all})}

\textbf{next}

\textbf{fix} \; x \; w \; i

\textbf{assume} \; a: \forall \; i. \; ?P \; i \; w \; \longrightarrow (\forall k. \; ?Q \; i \; k \; w)

\textbf{assume} \; b: \; i \leq \text{length} \; w

\textbf{show} \; 0 < i \; \longrightarrow (\forall k. \; \text{set(} \text{init'} \; G \; w \; (\text{Suc} \; k) \; (k + i, \; \text{Suc} \; 0)) = \text{CYK} \; G \; (x\#w) \; i \\\n\quad \quad \quad \quad \quad \quad (\text{Suc} \; 0))

\textbf{proof(clarify, case-tac i, simp-all, subst \text{CYK-egI[simplified], simp, erule subst, rule b, simp})}

\textbf{fix} \; k \; j
assume \( c \colon i = \text{Suc} \ j \)

note \( a \)[rule-format, where \( i=j \) and \( k=\text{Suc} \ k \)]

with \( b \) and \( c \) have set\((\text{init}' \ G \ w \, (\text{Suc} \ k) \, (\text{Suc} \ k + j, \text{Suc} \ 0))\) = CYK \( G \ w \ j \)

(Suc 0) by simp

also with \( b \) and \( c \) have ... = \{\( S, \ (\text{Leaf} \ (w ! j)) \) \( \in \) set \G\} by(subst CYK-eq1[simplified], simp-all)

finally show set\((\text{init}' \ G \ w \, (\text{Suc} \ k) \, (\text{Suc} \ (k + j), \text{Suc} \ 0))\) = \{\( S, \ (\text{Leaf} \ (w ! j)) \) \( \in \) set \G\} by simp

qed

with assms have \( \forall \) \( k \) . \(?Q i k w \) by simp

note this[rule-format, where \( k=0 \)]

thus \?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 \( \text{init} \colon ('n, 't) \text{CNG} \Rightarrow 't \text{list} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat} \Rightarrow 'n \text{list}) \text{ option} \)

where \( \text{init} \ G \ [] \ k = \text{None} \ |

\( \text{init} \ G \ [t] \ k = \text{case} \ (\text{init-match} \ G \ t) \ of \)

\( [] \Rightarrow \text{None} \ |
\)

\( \text{xs} \Rightarrow \text{Some}(\text{emptyT}((k, 1) := \text{xs}))) \) |

\( \text{init} \ G \ (t\#ts) \ k = \text{case} \ (\text{init-match} \ G \ t) \ of \)

\( [] \Rightarrow \text{None} \ |
\)

\( \text{xs} \Rightarrow \text{case} \ (\text{init} \ G \ ts \ (k + 1)) \) of \)

\( \text{None} \Rightarrow \text{None} \ |
\)

\( \text{Some} \ T \Rightarrow \text{Some}(T((k, 1) := \text{xs}))) \)

lemma \( \text{init1}[\text{rule-format}] : \)

\( \forall \ T . \text{init} \ G \ w \ k = \text{Some} \ T \longrightarrow \)

\( \text{init}' \ G \ w \ k = T \)

by(induct-tac \ G \ w \ k \ rule: \text{init.induct, clarsimp+}, \text{simp split: list.split-asm, rule ext, clarsimp+},

\text{simp split: list.split-asm option.split-asm, rule ext, clarsimp, force})

lemma \( \text{init2} : \)

\( (\text{init} \ G \ w \ k = \text{None}) = \)

\( (w = [] \lor (\exists a \in \text{set} \ w . \text{init-match} \ G \ a = [])) \)

by(induct-tac \ G \ w \ k \ rule: \text{init.induct, simp, simp split: list.split,}

\text{simp split: list.split option.split, force})

4.3 The overall procedure

definition \( \text{cyk} \ G \ S \ w = \text{case} \ \text{init} \ G \ w \ 0 \ \text{of} \)

\( \text{None} \Rightarrow \text{None} \ |
\)

\( \text{Some} \ T \Rightarrow \text{Some}(T((k, 1) := \text{xs}))) \)

22
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(simp add: cyk-def split: option.split, simp-all add: Let-def,
  rule conjI, subst init2, simp, rule conjI)
show w = [] ⇒ [] /∈ Lang G S by (clarify, drule Lang-no-Nil, clarify)
next
show (∃ x ∈ set w. init-match G x = []) ⇒ w /∈ Lang G S by (clarify, drule
Lang-term, subst (asm)init-match [THEN sym], force)
next
show ∀ T. init G w 0 = Some T ⇒
  ((length w = Suc 0 ⇒ S ∈ set(T(0, Suc 0))) ∧
   (length w ≠ Suc 0 ⇒ S ∈ set(main G T (length w) 0 2 (0, length w)))) =
   (w ∈ Lang G S) (is ∀ T. ?P T ⇒ ?L T = ?R)
proof clarify
fix T
assume a: ?P T
hence b: init' G w 0 = T by (rule init1)
note init2[THEN iffD2, OF disjI1]
have c: w ≠ [] by (clarify, drule init2[where G=G and k=0, THEN iffD2, OF
disjI1], simp add: a)
  have ?L (init' G w 0) = ?R
  proof(case-tac length w = 1, simp-all)
    assume d: length w = Suc 0
    show S ∈ set(init' G w 0 (0, Suc 0)) = ?R
      by (subst init'[simplified], simp add: d, subst CYK-Lang[THEN sym], simp add: d)
  next
  assume length w ≠ Suc 0
  with c have 1 < length w by (case-tac w, simp-all)
  hence d: Suc(Suc 0) ≤ length w by simp
  show (S ∈ set(main G (init' G w 0) (length w) 0 2 (0, length w))) = (w ∈ Lang G S)
  proof(subst main, simp-all, rule d)
    fix i' j'
    assume j' < 2 and Suc 0 ≤ j'
    hence e: j' = 1 by simp
    assume i' + j' ≤ length w
    with e have f: i' + 1 ≤ length w by simp
    have set(init' G w 0 (i', 1)) = CYK G w i' 1 by (rule init', rule f)
    with e show set(init' G w 0 (i', j')) = CYK G w i' j' by simp
  next
next

23
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","a","b","a"'],
    ["b","a","b","a"]
  end

References

[1] D. H. Younger. Recognition and parsing of context-free languages in