Greibach Normal Form

Alexander Haberl and Tobias Nipkow and Akihisa Yamada

September 12, 2025

Abstract

This theory formalizes Hopcroft and Ullman's algorithm [3] to transform a set of productions into Greibach Normal Form (GNF) [2]. We concentrate on the essential property of the GNF: every production starts with a terminal; the tail of a rhs may contain further terminals. The complexity of the algorithm can be exponential.

theory *Greibach_Normal_Form*
imports
Context_Free_Grammar.Context_Free_Grammar
Fresh_Identifiers.Fresh_Nat
begin

declare *relpowp.simps*(*2*)[*simp del*]

1 Aux Lemmas

lemma *Nts_mono*: $G \subseteq H \Longrightarrow Nts\ G \subseteq Nts\ H$
by (*auto simp add*: *Nts_def*)

lemma *derivern_prepend*: $R \vdash u \Rightarrow r(n)\ v \Longrightarrow R \vdash p\ @\ u \Rightarrow r(n)\ p\ @\ v$
by (*fastforce simp*: *derivern_iff_rev_deriveln rev_map deriveln_append rev_eq_append_conv*)

lemma *Lang_subset_if_Ders_subset*: $Ders\ R\ A \subseteq Ders\ R'\ A \Longrightarrow Lang\ R\ A \subseteq Lang\ R'\ A$
by (*auto simp add*: *Lang_def Ders_def*)

lemma *Eps_free_deriven_Nil*:
$[\![\ Eps\_free\ R;\ R \vdash l \Rightarrow(n)\ []\ ]\!] \Longrightarrow l = []$
by (*metis Eps_free_derives_Nil relpowp_imp_rtranclp*)

lemma *nts_syms_empty_iff*: $nts\_syms\ w = \{\} \longleftrightarrow (\exists u.\ w = map\ Tm\ u)$
by(*induction w*) (*auto simp*: *ex_map_conv split*: *sym.split*)

```
lemma non_word_has_last_Nt: nts_syms w ≠ {} ⟹ ∃u A v. w = u @ [Nt A]
@ map Tm v
proof (induction w)
  case Nil
  then show ?case by simp
next
  case (Cons a list)
  then show ?case using nts_syms_empty_iff[of list]
    by(auto simp: Cons_eq_append_conv split: sym.splits)
qed

lemma nts_syms_rev: nts_syms (rev w) = nts_syms w
by(auto simp: nts_syms_def)
```

Sentential form that is not a word has a first *Nt*.

```
lemma non_word_has_first_Nt: nts_syms w ≠ {} ⟹ ∃u A v. w = map Tm u
@ Nt A # v
  using nts_syms_rev non_word_has_last_Nt[of rev w]
 by (metis append.assoc append_Cons append_Nil rev.simps(2) rev_eq_append_conv
rev_map)
```

If there exists a derivation from u to v then there exists one which does not use productions of the form $A \to A$.

```
lemma no_self_loops_derivels: P ⊢ u ⇒l(n) v ⟹ {p∈P. ¬(∃A. p = (A,[Nt
A]))} ⊢ u ⇒l* v
proof(induction n arbitrary: u)
  case 0
  then show ?case by simp
next
  case (Suc n)
  then have ∃w. P ⊢ u ⇒l w ∧ P ⊢ w ⇒l(n) v
    by (simp add: relpowp_Suc_D2)
  then obtain w where W: P ⊢ u ⇒l w ∧ P ⊢ w ⇒l(n) v by blast
  then have ∃ (A,x) ∈ P. ∃u1 u2. u = map Tm u1 @ Nt A # u2 ∧ w = map
Tm u1 @ x @ u2
    by (simp add: derivel_iff)
  then obtain A x u1 u2 where prod: u = map Tm u1 @Nt A#u2 ∧ w = map
Tm u1 @x@u2 ∧ (A, x) ∈ P
    by blast
  then show ?case
  proof(cases x = [Nt A])
    case True
    then have u = w using prod by auto
    then show ?thesis using Suc W by auto
  next
    case False
    then have (A, x) ∈ {p∈P. ¬(∃A. p = (A,[Nt A]))} using prod by (auto)
    then show ?thesis using Suc W
      by (metis (lifting) converse_rtranclp_into_rtranclp derivel.intros prod)
```

  qed
qed

A decomposition of a derivation from a sentential form to a word into multiple derivations that derive words.

lemma *derivern_snoc_Nt_Tms_decomp1*:
  $R \vdash p\ @\ [Nt\ A] \Rightarrow r(n)\ map\ Tm\ q$
  $\Longrightarrow \exists pt\ At\ w\ k\ m.\ R \vdash p \Rightarrow(k)\ map\ Tm\ pt \wedge R \vdash w \Rightarrow(m)\ map\ Tm\ At \wedge (A, w) \in R$
  $\wedge\ q = pt\ @\ At \wedge n = Suc(k + m)$
proof −
  assume *assm*: $R \vdash p\ @\ [Nt\ A] \Rightarrow r(n)\ map\ Tm\ q$
  then have $R \vdash p\ @\ [Nt\ A] \Rightarrow(n)\ map\ Tm\ q$ **by** (*simp add*: *derivern_iff_deriven*)
  then have $\exists n1\ n2\ q1\ q2.\ n = n1 + n2 \wedge map\ Tm\ q = q1@q2 \wedge R \vdash p \Rightarrow(n1)\ q1 \wedge R \vdash [Nt\ A] \Rightarrow(n2)\ q2$
    using *deriven_append_decomp* **by** *blast*
  then obtain *n1 n2 q1 q2*
    where *decomp1*: $n = n1 + n2 \wedge map\ Tm\ q = q1\ @\ q2 \wedge R \vdash p \Rightarrow(n1)\ q1 \wedge R \vdash [Nt\ A] \Rightarrow(n2)\ q2$
    by *blast*
  then have $\exists pt\ At.\ q1 = map\ Tm\ pt \wedge q2 = map\ Tm\ At \wedge q = pt\ @\ At$
    by (*meson map_eq_append_conv*)
  then obtain *pt At* **where** *decomp_tms*: $q1 = map\ Tm\ pt \wedge q2 = map\ Tm\ At \wedge q = pt\ @\ At$ **by** *blast*
  then have $\exists w\ m.\ n2 = Suc\ m \wedge R \vdash w \Rightarrow(m)\ (map\ Tm\ At) \wedge (A,w) \in R$
    using *decomp1*
    by (*auto simp add*: *deriven_start1*)
  then obtain *w m* **where** $n2 = Suc\ m \wedge R \vdash w \Rightarrow(m)\ (map\ Tm\ At) \wedge (A,w) \in R$ **by** *blast*
  then have $R \vdash p \Rightarrow(n1)\ map\ Tm\ pt \wedge R \vdash w \Rightarrow(m)\ map\ Tm\ At \wedge (A, w) \in R$
    $\wedge\ q = pt\ @\ At \wedge n = Suc(n1 + m)$
    using *decomp1 decomp_tms* **by** *auto*
  then show *?thesis* **by** *blast*
qed

A decomposition of a derivation from a sentential form to a word into multiple derivations that derive words.

lemma *word_decomp1*:
  $R \vdash p\ @\ [Nt\ A]\ @\ map\ Tm\ ts \Rightarrow(n)\ map\ Tm\ q$
  $\Longrightarrow \exists pt\ At\ w\ k\ m.\ R \vdash p \Rightarrow(k)\ map\ Tm\ pt \wedge R \vdash w \Rightarrow(m)\ map\ Tm\ At \wedge (A, w) \in R$
  $\wedge\ q = pt\ @\ At\ @\ ts \wedge n = Suc(k + m)$
proof −
  assume *assm*: $R \vdash p\ @\ [Nt\ A]\ @\ map\ Tm\ ts \Rightarrow(n)\ map\ Tm\ q$
  then have $\exists q1\ q2\ n1\ n2.\ R \vdash p\ @\ [Nt\ A] \Rightarrow(n1)\ q1 \wedge R \vdash map\ Tm\ ts \Rightarrow(n2)\ q2$
    $\wedge\ map\ Tm\ q = q1\ @\ q2 \wedge n = n1 + n2$
    using *deriven_append_decomp*[*of n R p* @ [*Nt A*] *map Tm ts map Tm q*] **by** *auto*

then obtain *q1 q2 n1 n2*
where *P*: $R \vdash p@[Nt\ A] \Rightarrow(n1)\ q1 \wedge R \vdash map\ Tm\ ts \Rightarrow(n2)\ q2 \wedge map\ Tm\ q = q1\ @\ q2 \wedge n = n1 + n2$
by *blast*
then have $(\exists q1t\ q2t.\ q1 = map\ Tm\ q1t \wedge q2 = map\ Tm\ q2t \wedge q = q1t\ @\ q2t)$
using *deriven_from_TmsD map_eq_append_conv* **by** *blast*
then obtain *q1t q2t* **where** *P1*: $q1 = map\ Tm\ q1t \wedge q2 = map\ Tm\ q2t \wedge q = q1t\ @\ q2t$ **by** *blast*
then have $q2 = map\ Tm\ ts$ **using** *P*
using *deriven_from_TmsD* **by** *blast*
then have *1*: $ts = q2t$ **using** *P1*
by (*metis list.inj_map_strong sym.inject*(*2*))
then have $n1 = n$ **using** *P*
by (*metis add.right_neutral not_derive_from_Tms relpowp_E2*)
then have $\exists pt\ At\ w\ k\ m.\ R \vdash p \Rightarrow(k)\ map\ Tm\ pt \wedge R \vdash w \Rightarrow(m)\ map\ Tm\ At \wedge (A,\ w) \in R$
$\wedge\ q1t = pt\ @\ At \wedge n = Suc(k + m)$
using *P P1 derivern_snoc_Nt_Tms_decomp1*[*of n R p A q1t*] *derivern_iff_deriven* **by** *blast*
then obtain *pt At w k m* **where** *P2*: $R \vdash p \Rightarrow(k)\ map\ Tm\ pt \wedge R \vdash w \Rightarrow(m)\ map\ Tm\ At \wedge (A,\ w) \in R$
$\wedge\ q1t = pt\ @\ At \wedge n = Suc(k + m)$
by *blast*
then have $q = pt\ @\ At\ @\ ts$ **using** *P1 1* **by** *auto*
then show *?thesis* **using** *P2* **by** *blast*
qed

Sentential form that derives to terminals and has a Nt in it has a derivation that starts with some rule acting on that Nt.

lemma *deriven_start_sent*:
$R \vdash u\ @\ Nt\ V\ \#\ w \Rightarrow(Suc\ n)\ map\ Tm\ x \Longrightarrow \exists v.\ (V,\ v) \in R \wedge R \vdash u\ @\ v\ @\ w \Rightarrow(n)\ map\ Tm\ x$
proof −
assume *assm*: $R \vdash u\ @\ Nt\ V\ \#\ w \Rightarrow(Suc\ n)\ map\ Tm\ x$
then have $\exists n1\ n2\ xu\ xvw.\ Suc\ n = n1 + n2 \wedge map\ Tm\ x = xu\ @\ xvw \wedge R \vdash u \Rightarrow(n1)\ xu$
$\wedge\ R \vdash Nt\ V\ \#\ w \Rightarrow(n2)\ xvw$
by (*simp add*: *deriven_append_decomp*)
then obtain *n1 n2 xu xvw*
where *P1*: $Suc\ n = n1 + n2 \wedge map\ Tm\ x = xu\ @\ xvw \wedge R \vdash u \Rightarrow(n1)\ xu \wedge R \vdash Nt\ V\ \#\ w \Rightarrow(n2)\ xvw$
by *blast*
then have *t*: $\nexists t.\ xvw = Nt\ V\ \#\ t$
by (*metis append_eq_map_conv map_eq_Cons_D sym.distinct*(*1*))
then have $(\exists n3\ n4\ v\ xv\ xw.\ n2 = Suc\ (n3 + n4) \wedge xvw = xv\ @\ xw \wedge (V,v) \in R$
$\wedge\ R \vdash v \Rightarrow(n3)\ xv \wedge R \vdash w \Rightarrow(n4)\ xw)$
using *P1 t* **by** (*simp add*: *deriven_Cons_decomp*)
then obtain *n3 n4 v xv xw*

where *P2*: *n2* = *Suc* (*n3* + *n4*) ∧ *xvw* = *xv* @ *xw* ∧ (*V*,*v*) ∈ *R* ∧ *R* ⊢ *v* ⇒(*n3*) *xv* ∧ *R* ⊢ *w* ⇒(*n4*) *xw*
by *blast*
then have *R* ⊢ *v* @ *w* ⇒(*n3* + *n4*) *xvw* **using** *P2*
using *deriven_append_decomp diff_Suc_1* **by** *blast*
then have *R* ⊢ *u* @ *v* @ *w* ⇒(*n1* + *n3* + *n4*) *map Tm x* **using** *P1 P2 deriven_append_decomp*
using *ab_semigroup_add_class.add_ac*(*1*) **by** *blast*
then have *R* ⊢ *u* @ *v* @ *w* ⇒(*n*) *map Tm x* **using** *P1 P2*
by (*simp add*: *add.assoc*)
then show *?thesis* **using** *P2* **by** *blast*
qed

definition *nts_syms_list* :: (′*n*,′*t*)*syms* ⇒ ′*n list* ⇒ ′*n list* **where**
nts_syms_list sys = *foldr* (λ*sy ns*. *case sy of Nt A* ⇒ *List.insert A ns* | *Tm _* ⇒ *ns*) *sys*

definition *nts_prods_list* :: (′*n*,′*t*)*prods* ⇒ ′*n list* **where**
nts_prods_list ps = *foldr* (λ(*A*,*sys*) *ns*. *List.insert A* (*nts_syms_list sys ns*)) *ps* []

lemma *set_nts_syms_list*: *set*(*nts_syms_list sys ns*) = *nts_syms sys* ∪ *set ns*
unfolding *nts_syms_list_def*
by(*induction sys arbitrary*: *ns*) (*auto split*: *sym.split*)

lemma *set_nts_prods_list*: *set*(*nts_prods_list ps*) = *nts ps*
by(*induction ps*) (*auto simp*: *nts_prods_list_def Nts_def set_nts_syms_list split*: *prod.splits*)

lemma *distinct_nts_syms_list*: *distinct*(*nts_syms_list sys ns*) = *distinct ns*
unfolding *nts_syms_list_def*
by(*induction sys arbitrary*: *ns*) (*auto split*: *sym.split*)

lemma *distinct_nts_prods_list*: *distinct*(*nts_prods_list ps*)
by(*induction ps*) (*auto simp*: *nts_prods_list_def distinct_nts_syms_list split*: *sym.split*)

fun *freshs* :: (′*a*::*fresh*) *set* ⇒ ′*a list* ⇒ ′*a list* **where**
freshs X [] = [] |
freshs X (*a*#*as*) = (*let a*′ = *fresh X a in a*′ # *freshs* (*insert a*′ *X*) *as*)

lemma *length_freshs*: *finite X* ⟹ *length*(*freshs X as*) = *length as*
by(*induction as arbitrary*: *X*)(*auto simp*: *fresh_notIn Let_def*)

lemma *freshs_disj*: *finite X* ⟹ *X* ∩ *set*(*freshs X as*) = {}
proof(*induction as arbitrary*: *X*)
case *Cons*

then show *?case* **using** *fresh_notIn* **by**(*auto simp add*: *Let_def*)
qed *simp*

lemma *freshs_distinct*: *finite X* ⟹ *distinct* (*freshs X as*)
proof(*induction as arbitrary*: *X*)
case (*Cons a as*)
then show *?case*
using *freshs_disj*[*of insert* (*fresh X a*) *X as*] *fresh_notIn* **by**(*auto simp add*: *Let_def*)
qed *simp*

This theory formalizes a method to transform a set of productions into Greibach Normal Form (GNF) [2]. We concentrate on the essential property of the GNF: every production starts with a *Tm*; the tail of a rhs can contain further terminals. This is formalized as *GNF_hd* below. This more liberal definition of GNF is also found elsewhere [1].

The algorithm consists of two phases:

- *solve_tri* converts the productions into a *triangular* form, where Nt *Ai* does not depend on Nts *Ai*, . . ., *An*. This involves the elimination of left-recursion and is the heart of the algorithm.
- *expand_tri* expands the triangular form by substituting in: Due to triangular form, *A0* productions satisfy *GNF_hd* and we can substitute them into the heads of the remaining productions. Now all *A1* productions satisfy *GNF_hd*, and we continue until all productions satisfy *GNF_hd*.

This is essentially the algorithm given by Hopcroft and Ullman [3], except that we can drop the conversion to Chomsky Normal Form because of our more liberal *GNF_hd*.

2 Function Definitions

Depend on: *A* depends on *B* if there is a rule *A* → *B w*:

definition *dep_on* :: (′*n*,′*t*) *Prods* ⇒ ′*n* ⇒ ′*n set* **where**
dep_on R A = {*B*. ∃ *w*. (*A*,*Nt B* # *w*) ∈ *R*}

GNF property: All productions start with a terminal.

definition *GNF_hd* :: (′*n*,′*t*)*Prods* ⇒ *bool* **where**
GNF_hd R = (∀ (*A*, *w*) ∈ *R*. ∃ *t*. *hd w* = *Tm t*)

GNF property expressed via *dep_on*:

definition *GNF_hd_dep_on* :: (′*n*,′*t*)*Prods* ⇒ *bool* **where**
GNF_hd_dep_on R = (∀ *A* ∈ *Nts R*. *dep_on R A* = {})

abbreviation *lrec_Prods* :: $('n,'t)Prods \Rightarrow 'n \Rightarrow 'n\ set \Rightarrow ('n,'t)Prods$ **where**
$lrec\_Prods\ R\ A\ S \equiv \{(A',Bw) \in R.\ A'=A \wedge (\exists\, w\ B.\ Bw = Nt\ B \# w \wedge B \in S)\}$

abbreviation *subst_hd* :: $('n,'t)Prods \Rightarrow ('n,'t)Prods \Rightarrow 'n \Rightarrow ('n,'t)Prods$ **where**
$subst\_hd\ R\ X\ A \equiv \{(A,v@w)\ |v\ w.\ \exists\, B.\ (A,Nt\ B \# w) \in X \wedge (B,v) \in R\}$

Expand head: Replace all rules $A \to B\ w$ where $B \in Ss$ (Ss = solved Nts in *triangular* form) by $A \to v\ w$ where $B \to v$. Starting from the end of Ss.

fun *expand_hd* :: $'n \Rightarrow 'n\ list \Rightarrow ('n,'t)Prods \Rightarrow ('n,'t)Prods$ **where**
expand_hd A [] *R* = *R* |
expand_hd A (*S#Ss*) *R* =
(*let R′* = *expand_hd A Ss R*;
X = *lrec_Prods R′ A* {*S*};
Y = *subst_hd R′ X A*
in R′ − *X* ∪ *Y*)

lemma *Rhss_code*[*code*]: *Rhss P A* = *snd* ‘ {*Aw* ∈ *P*. *fst Aw* = *A*}
by(*auto simp add*: *Rhss_def image_iff*)

declare *expand_hd.simps*(*1*)[*code*]
lemma *expand_hd_Cons_code*[*code*]: *expand_hd A* (*S#Ss*) *R* =
(*let R′* = *expand_hd A Ss R*;
X = {*w* ∈ *Rhss R′ A*. *w* ≠ [] ∧ *hd w* = *Nt S*};
Y = (⋃(*B*,*v*) ∈ *R′*. ⋃*w* ∈ *X*. *if hd w* ≠ *Nt B then* {} *else* {(*A*,*v* @ *tl w*)})
in R′ − ({*A*} × *X*) ∪ *Y*)
by(*simp add*: *Rhss_def Let_def neq_Nil_conv Ball_def hd_append split*: *if_splits*, *safe*, *force*+)

Remove left-recursions: Remove left-recursive rules $A \to A\ w$:

definition *rm_lrec* :: $'n \Rightarrow ('n,'t)Prods \Rightarrow ('n,'t)Prods$ **where**
$rm\_lrec\ A\ R = R - \{(A,Nt\ A \# v)|v.\ True\}$

lemma *rm_lrec_code*[*code*]:
$rm\_lrec\ A\ R = \{Aw \in R.\ let\ (A',w) = Aw\ in\ A' \neq A \vee w = [] \vee hd\ w \neq Nt\ A\}$
by(*auto simp add*: *rm_lrec_def neq_Nil_conv*)

Make right-recursion of left-recursion: Conversion from left-recursion to right-recursion: Split A-rules into $A \to u$ and $A \to A\ v$. Keep $A \to u$ but replace $A \to A\ v$ by $A \to u\ A'$, $A' \to v$, $A' \to v\ A'$.

The then part of the if statement is only an optimisation, so that we do not introduce the $A \to u\ A'$ rules if we do not introduce any A' rules, but the function also works, if we always enter the else part.

definition *rrec_of_lrec* :: $'n \Rightarrow 'n \Rightarrow ('n,'t)Prods \Rightarrow ('n,'t)Prods$ **where**
rrec_of_lrec A A′ R =
(*let V* = {*v*. (*A*,*Nt A* # *v*) ∈ *R* ∧ *v* ≠ []};
U = {*u*. (*A*,*u*) ∈ *R* ∧ ¬(∃ *v*. *u* = *Nt A* # *v*) }

in if $V = \{\}$ *then* $R - \{(A, [Nt\ A])\}$ *else* $(\{A\} \times U) \cup (\bigcup u{\in}U.\ \{(A,u@[Nt\ A'])\}) \cup (\{A'\} \times V) \cup (\bigcup v{\in}V.\ \{(A',v\ @\ [Nt\ A'])\}))$

lemma *rrec_of_lrec_code*[*code*]: *rrec_of_lrec A A' R* =
(*let RA* = *Rhss R A*;
$V = tl$ ' $\{w \in RA.\ w \neq [] \wedge hd\ w = Nt\ A \wedge tl\ w \neq []\}$;
$U = \{u \in RA.\ u = [] \vee hd\ u \neq Nt\ A\ \}$
in if $V = \{\}$ *then* $R - \{(A, [Nt\ A])\}$ *else* $(\{A\} \times U) \cup (\bigcup u{\in}U.\ \{(A,u@[Nt\ A'])\}) \cup (\{A'\} \times V) \cup (\bigcup v{\in}V.\ \{(A',v\ @\ [Nt\ A'])\}))$
proof –
let *?RA* = *Rhss R A*
let $?Vc = tl$ ' $\{w \in ?RA.\ w \neq [] \wedge hd\ w = Nt\ A \wedge tl\ w \neq []\}$
let $?Uc = \{u \in ?RA.\ u = [] \vee hd\ u \neq Nt\ A\ \}$

let $?V = \{v.\ (A,Nt\ A\ \#\ v) \in R \wedge v \neq []\}$
let $?U = \{u.\ (A,u) \in R \wedge \neg(\exists v.\ u = Nt\ A\ \#\ v)\ \}$

have *1*: *?V* = *?Vc* **by** (*auto simp add*: *Rhss_def neq_Nil_conv image_def*)
moreover have *2*: *?U* = *?Uc* **by** (*auto simp add*: *Rhss_def neq_Nil_conv*)

ultimately show *?thesis*
unfolding *rrec_of_lrec_def Let_def* **by** *presburger*
qed

Solve left-recursions: Solves the left-recursion of Nt A by replacing it with a right-recursion on a fresh Nt A'. The fresh Nt A' is also given as a parameter.

definition *solve_lrec* :: *'n* ⇒ *'n* ⇒ (*'n*,*'t*)*Prods* ⇒ (*'n*,*'t*)*Prods* **where**
solve_lrec A A' R = *rm_lrec A R* ∪ *rrec_of_lrec A A' R*

lemmas *solve_lrec_defs* = *solve_lrec_def rm_lrec_def rrec_of_lrec_def Let_def Nts_def*

Solve triangular: Put R into triangular form wrt *As* (using the new Nts *As'*). In each step *A#As*, first the remaining Nts in *As* are solved, then A is solved. This should mean that in the result of the outermost *expand_hd A As*, A only depends on A. Then the A rules in the result of *solve_lrec A A'* are already in GNF. More precisely: the result should be in *triangular* form.

fun *solve_tri* :: *'a list* ⇒ *'a list* ⇒ (*'a*, *'b*) *Prods* ⇒ (*'a*, *'b*) *Prods* **where**
solve_tri [] _ *R* = *R* |
solve_tri (*A#As*) (*A'#As'*) *R* = *solve_lrec A A'* (*expand_hd A As* (*solve_tri As As' R*))

Triangular form wrt [*A1*,...,*An*] means that *Ai* must not depend on *Ai*, ..., *An*. In particular: *A0* does not depend on any *Ai*, its rules are already in GNF. Therefore one can convert a *triangular* form into GNF by backwards substitution: The rules for *Ai* are used to expand the heads of all *A*(*i*+*1*),...,*An* rules, starting with *A0*.

fun *triangular* :: *'n list* ⇒ (*'n* × (*'n*, *'t*) *sym list*) *set* ⇒ *bool* **where**

triangular [] *R* = *True* |
triangular (*A*#*As*) *R* = (*dep_on R A* ∩ ({*A*} ∪ *set As*) = {} ∧ *triangular As R*)

Remove self loops: Removes all productions of the form $A \rightarrow A$.

definition *rm_self_loops* :: ('*n*,'*t*) *Prods* ⇒ ('*n*,'*t*) *Prods* **where**
  rm_self_loops P = *P* − {*x*∈*P*. ∃ *A*. *x* = (*A*, [*Nt A*])}

Expand triangular: Expands all head-Nts of productions with a Lhs in *As* (*triangular* (*rev As*)). In each step *A*#*As* first all Nts in *As* are expanded, then every rule $A \rightarrow B\ w$ is expanded if $B \in$ *set As*. If the productions were in *triangular* form wrt *rev As* then *Ai* only depends on *A*(*i*+*1*), . . ., *An* which have already been expanded in the first part of the step and are in GNF. Then the all *A*-productions are also is in GNF after expansion.

fun *expand_tri* :: '*n list* ⇒ ('*n*,'*t*)*Prods* ⇒ ('*n*,'*t*)*Prods* **where**
expand_tri [] *R* = *R* |
expand_tri (*A*#*As*) *R* =
 (*let R*′ = *expand_tri As R*;
      X = *lrec_Prods R*′ *A* (*set As*);
      Y = *subst_hd R*′ *X A*
  in R′ − *X* ∪ *Y*)

declare *expand_tri.simps*(*1*)[*code*]
lemma *expand_tri_Cons_code*[*code*]: *expand_tri* (*S*#*Ss*) *R* =
 (*let R*′ = *expand_tri Ss R*;
      X = {*w* ∈ *Rhss R*′ *S*. *w* ≠ [] ∧ *hd w* ∈ *Nt* ‘ (*set Ss*)};
      Y = (⋃(*B*,*v*) ∈ *R*′. ⋃*w* ∈ *X*. *if hd w* ≠ *Nt B then* {} *else* {(*S*,*v* @ *tl w*)})
  in R′ − ({*S*} × *X*) ∪ *Y*)
by(*simp add*: *Let_def Rhss_def neq_Nil_conv Ball_def*, *safe*, *force*+)

The main function *gnf_hd* converts into *GNF_hd*:

definition *gnf_hd* :: ('*n*::*fresh*,'*t*)*prods* ⇒ ('*n*,'*t*)*Prods* **where**
gnf_hd ps =
  (*let As* = *nts_prods_list ps*;
       As′ = *freshs* (*set As*) *As*
   in expand_tri (*As*′ @ *rev As*) (*solve_tri As As*′ (*set ps*)))

3 Some Basic Lemmas

3.1 *Eps_free* preservation

lemma *Eps_free_expand_hd*: *Eps_free R* ⟹ *Eps_free* (*expand_hd A Ss R*)
  by (*induction A Ss R rule*: *expand_hd.induct*)
    (*auto simp add*: *Eps_free_def Let_def*)

lemma *Eps_free_solve_lrec*: *Eps_free R* ⟹ *Eps_free* (*solve_lrec A A*′ *R*)
  unfolding *solve_lrec_defs Eps_free_def* **by** (*auto*)

lemma *Eps_free_solve_tri*: *Eps_free R* ⟹ *length As* ≤ *length As′* ⟹ *Eps_free* (*solve_tri As As′ R*)
by (*induction As As′ R rule*: *solve_tri.induct*)
(*auto simp add*: *Eps_free_solve_lrec Eps_free_expand_hd*)

lemma *Eps_free_expand_tri*: *Eps_free R* ⟹ *Eps_free* (*expand_tri As R*)
by (*induction As R rule*: *expand_tri.induct*) (*auto simp add*: *Let_def Eps_free_def*)

3.2 Lemmas about *Nts* and *dep_on*

lemma *dep_on_Un*[*simp*]: *dep_on* (*R* ∪ *S*) *A* = *dep_on R A* ∪ *dep_on S A*
by(*auto simp add*: *dep_on_def*)

lemma *expand_hd_preserves_neq*: *B* ≠ *A* ⟹ (*B*,*w*) ∈ *expand_hd A Ss R* ⟷ (*B*,*w*) ∈ *R*
by(*induction A Ss R rule*: *expand_hd.induct*) (*auto simp add*: *Let_def*)

Let *R* be epsilon-free and in *triangular* form wrt *Bs*. After *expand_hd A Bs R*, *A* depends only on what *A* depended on before or what one of the *B* ∈ *Bs* depends on, but *A* does not depend on the *Bs*:

lemma *dep_on_expand_hd*:
⟦ *Eps_free R*; *triangular Bs R*; *distinct Bs*; *A* ∉ *set Bs* ⟧
⟹ *dep_on* (*expand_hd A Bs R*) *A* ⊆ (*dep_on R A* ∪ (⋃*B*∈*set Bs*. *dep_on R B*)) − *set Bs*
proof(*induction A Bs R rule*: *expand_hd.induct*)
case (*1 A R*)
then show *?case* **by** *simp*
next
case (*2 A B Bs R*)
then show *?case*
by(*fastforce simp add*: *Let_def dep_on_def Cons_eq_append_conv Eps_free_expand_hd Eps_free_Nil*
expand_hd_preserves_neq set_eq_iff)
qed

lemma *dep_on_subs_Nts*: *dep_on R A* ⊆ *Nts R*
by (*auto simp add*: *Nts_def dep_on_def*)

lemma *Nts_expand_hd_sub*: *Nts* (*expand_hd A As R*) ⊆ *Nts R*
proof (*induction A As R rule*: *expand_hd.induct*)
case (*1 A R*)
then show *?case* **by** *simp*
next
case (*2 A S Ss R*)
let *?R′* = *expand_hd A Ss R*
let *?X* = {(*Al*, *Bw*). (*Al*, *Bw*) ∈ *?R′* ∧ *Al* = *A* ∧ (∃ *w*. *Bw* = *Nt S* # *w*)}
let *?Y* = {(*A*, *v* @ *w*) |*v w*. (*A*, *Nt S* # *w*) ∈ *?R′* ∧ (*S*, *v*) ∈ *?R′*}

have *lhs_sub*: *Lhss ?Y* ⊆ *Lhss ?R′* **by** (*auto simp add*: *Lhss_def*)

have *B* ∉ *Rhs_Nts ?R′* ⟶ *B* ∉ *Rhs_Nts ?Y* **for** *B*
by (*fastforce simp add*: *Rhs_Nts_def split*: *prod.splits*)
then have *B* ∈ *Rhs_Nts ?Y* ⟶ *B* ∈ *Rhs_Nts ?R′* **for** *B* **by** *blast*
then have *rhs_sub*: *Rhs_Nts ?Y* ⊆ *Rhs_Nts ?R′* **by** *auto*

have *Nts ?Y* ⊆ *Nts ?R′* **using** *lhs_sub rhs_sub* **by** (*auto simp add*: *Nts_Lhss_Rhs_Nts*)
then have *Nts ?Y* ⊆ *Nts R* **using** *2* **by** *auto*
then show *?case* **using** *Nts_mono*[*of ?R′* − *?X*] *2* **by** (*auto simp add*: *Let_def Nts_Un*)
qed

lemma *Nts_solve_lrec_sub*: *Nts* (*solve_lrec A A′ R*) ⊆ *Nts R* ∪ {*A′*}
proof −
have *1*: *Nts* (*rm_lrec A R*) ⊆ *Nts R*
by (*auto simp add*: *Nts_mono rm_lrec_def*)

have *2*: *Lhss* (*rrec_of_lrec A A′ R*) ⊆ *Lhss R* ∪ {*A′*}
by (*auto simp add*: *rrec_of_lrec_def Let_def Lhss_def*)
have *3*: *Rhs_Nts* (*rrec_of_lrec A A′ R*) ⊆ *Rhs_Nts R* ∪ {*A′*}
by (*auto simp add*: *rrec_of_lrec_def Let_def Rhs_Nts_def*)

have *Nts* (*rrec_of_lrec A A′ R*) ⊆ *Nts R* ∪ {*A′*} **using** *2 3* **by** (*auto simp add*: *Nts_Lhss_Rhs_Nts*)
then show *?thesis* **using** *1* **by** (*auto simp add*: *solve_lrec_def Nts_Un*)
qed

lemma *Nts_solve_tri_sub*: *length As* ≤ *length As′* ⟹ *Nts* (*solve_tri As As′ R*) ⊆ *Nts R* ∪ *set As′*
proof (*induction As As′ R rule*: *solve_tri.induct*)
case (*1 uu R*)
then show *?case* **by** *simp*
next
case (*2 A As A′ As′ R*)
have *Nts* (*solve_tri* (*A* # *As*) (*A′* # *As′*) *R*) =
Nts (*solve_lrec A A′* (*expand_hd A As* (*solve_tri As As′ R*))) **by** *simp*
also have ... ⊆ *Nts* (*expand_hd A As* (*solve_tri As As′ R*)) ∪ {*A′*}
using *Nts_solve_lrec_sub*[*of A A′ expand_hd A As* (*solve_tri As As′ R*)] **by** *simp*
also have ... ⊆ *Nts* (*solve_tri As As′ R*) ∪ {*A′*}
using *Nts_expand_hd_sub*[*of A As solve_tri As As′ R*] **by** *auto*
finally show *?case* **using** *2* **by** *auto*
next
case (*3 v va c*)
then show *?case* **by** *simp*
qed

3.3 Lemmas about *triangular*

lemma *tri_Snoc_impl_tri*: *triangular* (*As* @ [*A*]) *R* $\Longrightarrow$ *triangular As R*
proof(*induction As R rule*: *triangular.induct*)
  case (*1 R*)
  then show *?case* **by** *simp*
next
  case (*2 A As R*)
  then show *?case* **by** *simp*
qed

If two parts of the productions are *triangular* and no Nts from the first part depend on ones of the second they are also *triangular* when put together.

lemma *triangular_append*:
  ⟦*triangular As R*; *triangular Bs R*; ∀ *A*∈*set As*. *dep_on R A* ∩ *set Bs* = {}⟧
  $\Longrightarrow$ *triangular* (*As*@*Bs*) *R*
  by (*induction As*) *auto*

4 Function *solve_tri*: Remove Left-Recursion and Convert into Triangular Form

4.1 Basic Lemmas

Mostly about rule inclusions in *solve_lrec*.

lemma *solve_lrec_rule_simp1*: *A* ≠ *B* $\Longrightarrow$ *A* ≠ *B'* $\Longrightarrow$ (*A*, *w*) ∈ *solve_lrec B B'* *R* $\longleftrightarrow$ (*A*, *w*) ∈ *R*
  unfolding *solve_lrec_defs* **by** (*auto*)

lemma *solve_lrec_rule_simp3*: *A* ≠ *A'* $\Longrightarrow$ *A'* ∉ *Nts R* $\Longrightarrow$ *Eps_free R*
  $\Longrightarrow$ (*A*, [*Nt A'*]) ∉ *solve_lrec A A' R*
  unfolding *solve_lrec_defs* **by** (*auto simp*: *Eps_free_def*)

lemma *solve_lrec_rule_simp7*: *A'* ≠ *A* $\Longrightarrow$ *A'* ∉ *Nts R* $\Longrightarrow$ (*A'*, *Nt A'* # *w*) ∉ *solve_lrec A A' R*
unfolding *solve_lrec_defs* **by**(*auto simp*: *neq_Nil_conv split*: *prod.splits*)

lemma *solve_lrec_rule_simp8*: *A'* ∉ *Nts R* $\Longrightarrow$ *B* ≠ *A'* $\Longrightarrow$ *B* ≠ *A*
  $\Longrightarrow$ (*B*, *Nt A'* # *w*) ∉ *solve_lrec A A' R*
unfolding *solve_lrec_defs* **by** (*auto split*: *prod.splits*)

lemma *dep_on_expand_hd_simp2*: *B* ≠ *A* $\Longrightarrow$ *dep_on* (*expand_hd A As R*) *B* = *dep_on R B*
  by (*auto simp add*: *dep_on_def expand_hd_preserves_neq*)

lemma *dep_on_solve_lrec_simp2*: *A* ≠ *B* $\Longrightarrow$ *A'* ≠ *B* $\Longrightarrow$ *dep_on* (*solve_lrec A A' R*) *B* = *dep_on R B*
unfolding *solve_lrec_defs dep_on_def* **by** (*auto*)

4.2 Triangular Form

expand_hd preserves triangular, if it does not expand a Nt considered in triangular.

lemma triangular_expand_hd: ⟦A ∉ set As; triangular As R⟧ ⟹ triangular As (expand_hd A Bs R)
  by (induction As) (auto simp add: dep_on_expand_hd_simp2)

Solving a Nt not considered by triangular preserves the triangular property.

lemma triangular_solve_lrec: ⟦A ∉ set As; A′ ∉ set As; triangular As R⟧
  ⟹ triangular As (solve_lrec A A′ R)
proof(induction As)
  case Nil
  then show ?case **by** simp
next
  case (Cons a As)
  have triangular (a # As) (solve_lrec A A′ R) =
   (dep_on (solve_lrec A A′ R) a ∩ ({a} ∪ set As) = {} ∧ triangular As (solve_lrec A A′ R))
    by simp
  also have ... = (dep_on (solve_lrec A A′ R) a ∩ ({a} ∪ set As) = {}) **using** Cons **by** auto
  also have ... = (dep_on R a ∩ ({a} ∪ set As) = {}) **using** Cons dep_on_solve_lrec_simp2
    by (metis list.set_intros(1))
  then show ?case **using** Cons **by** auto
qed

Solving more Nts does not remove the triangular property of previously solved Nts.

lemma part_triangular_induct_step:
  ⟦Eps_free R; distinct ((A#As)@(A′#As′)); triangular As (solve_tri As As′ R)⟧
  ⟹ triangular As (solve_tri (A#As) (A′#As′) R)
  by (cases As = [])
   (auto simp add: triangular_expand_hd triangular_solve_lrec)

Couple of small lemmas about dep_on and the solving of left-recursion.

lemma rm_lrec_rem_own_dep: A ∉ dep_on (rm_lrec A R) A
  by (auto simp add: dep_on_def rm_lrec_def)

lemma rrec_of_lrec_has_no_own_dep: A ≠ A′ ⟹ A ∉ dep_on (rrec_of_lrec A A′ R) A
  by (auto simp add: dep_on_def rrec_of_lrec_def Let_def Cons_eq_append_conv)

lemma solve_lrec_no_own_dep: A ≠ A′ ⟹ A ∉ dep_on (solve_lrec A A′ R) A
  by (auto simp add: solve_lrec_def rm_lrec_rem_own_dep rrec_of_lrec_has_no_own_dep)

lemma solve_lrec_no_new_own_dep: A ≠ A′ ⟹ A′ ∉ Nts R ⟹ A′ ∉ dep_on (solve_lrec A A′ R) A′

by (*auto simp add*: *dep_on_def solve_lrec_rule_simp7*)

lemma *dep_on_rem_lrec_simp*: *dep_on* (*rm_lrec A R*) *A* = *dep_on R A* − {*A*}
by (*auto simp add*: *dep_on_def rm_lrec_def*)

lemma *dep_on_rrec_of_lrec_simp*:
Eps_free R ⟹ *A* ≠ *A′* ⟹ *dep_on* (*rrec_of_lrec A A′ R*) *A* = *dep_on R A* − {*A*}
using *Eps_freeE_Cons*[*of R A* []]
by (*auto simp add*: *dep_on_def rrec_of_lrec_def Let_def Cons_eq_append_conv*)

lemma *dep_on_solve_lrec_simp*:
⟦*Eps_free R*; *A* ≠ *A′*⟧ ⟹ *dep_on* (*solve_lrec A A′ R*) *A* = *dep_on R A* − {*A*}
by (*simp add*: *dep_on_rem_lrec_simp dep_on_rrec_of_lrec_simp solve_lrec_def*)

lemma *dep_on_solve_tri_simp*: *B* ∉ *set As* ⟹ *B* ∉ *set As′* ⟹ *length As* ≤ *length As′*
⟹ *dep_on* (*solve_tri As As′ R*) *B* = *dep_on R B*
proof (*induction As As′ R rule*: *solve_tri.induct*)
case (*1 uu R*)
then show *?case* **by** *simp*
next
case (*2 A As A′ As′ R*)
have *dep_on* (*solve_tri* (*A#As*) (*A′#As′*) *R*) *B* = *dep_on* (*expand_hd A As* (*solve_tri As As′ R*)) *B*
using *2* **by** (*auto simp add*: *dep_on_solve_lrec_simp2*)
then show *?case* **using** *2* **by** (*auto simp add*: *dep_on_expand_hd_simp2*)
next
case (*3 v va c*)
then show *?case* **by** *simp*
qed

Induction step for showing that *solve_tri* removes dependencies of previously solved Nts.

lemma *triangular_dep_on_induct_step*:
assumes *Eps_free R length As* ≤ *length As′ distinct* ((*A#As*)@*A′#As′*) *triangular As* (*solve_tri As As′ R*)
shows *dep_on* (*solve_tri* (*A* # *As*) (*A′* # *As′*) *R*) *A* ∩ ({*A*} ∪ *set As*) = {}
proof(*cases As* = [])
case *True*
with *assms solve_lrec_no_own_dep* **show** *?thesis* **by** *fastforce*
next
case *False*
have *Eps_free* (*solve_tri As As′ R*)
using *assms Eps_free_solve_tri* **by** *auto*
then have *test*: *X* ∈ *set As* ⟹ *X* ∉ *dep_on* (*expand_hd A As* (*solve_tri As As′ R*)) *A* **for** *X*
using *assms dep_on_expand_hd*
by (*metis distinct.simps*(*2*) *distinct_append insert_Diff subset_Diff_insert*)

have *A*: *triangular As* (*solve_tri* (*A* # *As*) (*A′* # *As′*) *R*)
using *part_triangular_induct_step assms* **by** *metis*

have *dep_on* (*solve_tri* (*A* # *As*) (*A′* # *As′*) *R*) *A* ∩ ({*A*} ∪ *set As*)
= (*dep_on* (*expand_hd A As* (*solve_tri As As′ R*)) *A* − {*A*}) ∩ ({*A*} ∪ *set*
As)
using *assms* **by** (*simp add*: *dep_on_solve_lrec_simp Eps_free_solve_tri Eps_free_expand_hd*)
also have ... = *dep_on* (*expand_hd A As* (*solve_tri As As′ R*)) *A* ∩ *set As*
using *assms* **by** *auto*
also have ... = {} **using** *test* **by** *fastforce*
finally show *?thesis* **by** *auto*
qed

theorem *triangular_solve_tri*: ⟦ *Eps_free R*; *length As* ≤ *length As′*; *distinct*(*As*
@ *As′*)⟧
⟹ *triangular As* (*solve_tri As As′ R*)
proof(*induction As As′ R rule*: *solve_tri.induct*)
case (*1 uu R*)
then show *?case* **by** *simp*
next
case (*2 A As A′ As′ R*)
then have *length As* ≤ *length As′* ∧ *distinct* (*As* @ *As′*) **by** *auto*
then have *A*: *triangular As* (*solve_tri* (*A* # *As*) (*A′* # *As′*) *R*)
using *part_triangular_induct_step 2 2.IH* **by** *metis*

have (*dep_on* (*solve_tri* (*A* # *As*) (*A′* # *As′*) *R*) *A* ∩ ({*A*} ∪ *set As*) = {})
using *triangular_dep_on_induct_step 2*
by (*metis* ‹*length As* ≤ *length As′* ∧ *distinct* (*As* @ *As′*)›)
then show *?case* **using** *A* **by** *simp*
next
case (*3 v va c*)
then show *?case* **by** *simp*
qed

lemma *dep_on_solve_tri_Nts_R*:
⟦*Eps_free R*; *B* ∈ *set As*; *distinct* (*As* @ *As′*); *set As′* ∩ *Nts R* = {}; *length As*
≤ *length As′*⟧
⟹ *dep_on* (*solve_tri As As′ R*) *B* ⊆ *Nts R*
proof (*induction As As′ R arbitrary*: *B rule*: *solve_tri.induct*)
case (*1 uu R*)
then show *?case* **by** (*simp add*: *dep_on_subs_Nts*)
next
case (*2 A As A′ As′ R*)
then have *F1*: *dep_on* (*solve_tri As As′ R*) *B* ⊆ *Nts R*
by (*cases B* = *A*) (*simp_all add*: *dep_on_solve_tri_simp dep_on_subs_Nts*)
then have *F2*: *dep_on* (*expand_hd A As* (*solve_tri As As′ R*)) *B* ⊆ *Nts R*
proof (*cases B* = *A*)
case *True*

have *triangular As* (*solve_tri As As′ R*) **using** *2* **by** (*auto simp add*: *triangular_solve_tri*)
then have *dep_on* (*expand_hd A As* (*solve_tri As As′ R*)) *B* ⊆ *dep_on* (*solve_tri As As′ R*) *B*
∪ ⋃ (*dep_on* (*solve_tri As As′ R*) ‘ *set As*) − *set As*
using *2 True* **by** (*auto simp add*: *dep_on_expand_hd Eps_free_solve_tri*)
also have ... ⊆ *Nts R* **using** *2.IH 2 F1* **by** *auto*
finally show *?thesis*.
next
case *False*
then show *?thesis* **using** *F1* **by** (*auto simp add*: *dep_on_expand_hd_simp2*)
qed
then have *dep_on* (*solve_lrec A A′* (*expand_hd A As* (*solve_tri As As′ R*))) *B* ⊆ *Nts R*
proof (*cases B* = *A*)
case *True*
then show *?thesis*
using *2 F2* **by** (*auto simp add*: *dep_on_solve_lrec_simp Eps_free_solve_tri Eps_free_expand_hd*)
next
case *False*
have *B* ≠ *A′* **using** *2* **by** *auto*
then show *?thesis* **using** *2 F2 False* **by** (*simp add*: *dep_on_solve_lrec_simp2*)
qed
then show *?case* **by** *simp*
next
case (*3 v va c*)
then show *?case* **by** *simp*
qed

lemma *triangular_unused_Nts*: *set As* ∩ *Nts R* = {} ⟹ *triangular As R*
proof (*induction As*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a As*)
have *dep_on R a* ⊆ *Nts R* **by** (*simp add*: *dep_on_subs_Nts*)
then have *dep_on R a* ∩ (*set As* ∪ {*a*}) = {} **using** *Cons* **by** *auto*
then show *?case* **using** *Cons* **by** *auto*
qed

The newly added Nts in *solve_lrec* are in *triangular* form wrt *rev As′*.

lemma *triangular_rev_As′_solve_tri*:
⟦*set As′* ∩ *Nts R* = {}; *distinct* (*As* @ *As′*); *length As* ≤ *length As′*⟧
⟹ *triangular* (*rev As′*) (*solve_tri As As′ R*)
proof (*induction As As′ R rule*: *solve_tri.induct*)
case (*1 uu R*)
then show *?case* **by** (*auto simp add*: *triangular_unused_Nts*)
next

case (*2 A As A′ As′ R*)
then have *triangular* (*rev As′*) (*solve_tri As As′ R*) **by** *simp*
then have *triangular* (*rev As′*) (*expand_hd A As* (*solve_tri As As′ R*))
using *2* **by** (*auto simp add*: *triangular_expand_hd*)
then have *F1*: *triangular* (*rev As′*) (*solve_tri* (*A#As*) (*A′#As′*) *R*)
using *2* **by** (*auto simp add*: *triangular_solve_lrec*)
have *Nts* (*solve_tri As As′ R*) ⊆ *Nts R* ∪ *set As′* **using** *2* **by** (*auto simp add*: *Nts_solve_tri_sub*)
then have *F_nts*: *Nts* (*expand_hd A As* (*solve_tri As As′ R*)) ⊆ *Nts R* ∪ *set As′*
using *Nts_expand_hd_sub*[*of A As* (*solve_tri As As′ R*)] **by** *auto*
then have *A′* ∉ *dep_on* (*solve_lrec A A′* (*expand_hd A As* (*solve_tri As As′ R*))) *A′*
using *2 solve_lrec_no_new_own_dep*[*of A A′*] **by** *auto*
then have *F2*: *triangular* [*A′*] (*solve_tri* (*A#As*) (*A′#As′*) *R*) **by** *auto*
have ∀ *a*∈*set As′*. *dep_on* (*solve_tri* (*A#As*) (*A′#As′*) *R*) *a* ∩ *set* [*A′*] = {}
proof
fix *a*
assume *a* ∈ *set As′*
then have *A′* ∉ *Nts* (*expand_hd A As* (*solve_tri As As′ R*)) ∧ *a* ≠ *A* **using** *F_nts 2* **by** *auto*
then show *dep_on* (*solve_tri* (*A#As*) (*A′#As′*) *R*) *a* ∩ *set* [*A′*] = {}
using *2 solve_lrec_rule_simp8*[*of A′* (*expand_hd A As* (*solve_tri As As′ R*)) *a A*]
solve_lrec_rule_simp7[*of A′*]
by (*cases a* = *A′*) (*auto simp add*: *dep_on_def*)
qed

then have *triangular* (*rev* (*A′#As′*)) (*solve_tri* (*A#As*) (*A′#As′*) *R*)
using *F1 F2* **by** (*auto simp add*: *triangular_append*)
then show *?case* **by** *auto*
next
case (*3 v va c*)
then show *?case* **by** *auto*
qed

The entire set of productions is in *triangular* form after *solve_tri* wrt *As*@(*rev As′*).

theorem *triangular_As_As′_solve_tri*:
assumes *Eps_free R length As* ≤ *length As′ distinct*(*As* @ *As′*) *Nts R* ⊆ *set As*
shows *triangular* (*As*@(*rev As′*)) (*solve_tri As As′ R*)
proof −
from *assms* **have** *1*: *triangular As* (*solve_tri As As′ R*) **by** (*auto simp add*: *triangular_solve_tri*)
have *set As′* ∩ *Nts R* = {} **using** *assms* **by** *auto*
then have *2*: *triangular* (*rev As′*) (*solve_tri As As′ R*)
using *assms* **by** (*auto simp add*: *triangular_rev_As′_solve_tri*)
have *set As′* ∩ *Nts R* = {} **using** *assms* **by** *auto*
then have ∀ *A*∈*set As*. *dep_on* (*solve_tri As As′ R*) *A* ⊆ *Nts R*

using *assms* **by** (*auto simp add*: *dep_on_solve_tri_Nts_R*)
then have $\forall A{\in}set\ As.\ dep\_on\ (solve\_tri\ As\ As'\ R)\ A \cap set\ As' = \{\}$ **using** *assms* **by** *auto*
then show *?thesis* **using** *1 2* **by** (*auto simp add*: *triangular_append*)
qed

4.3 *solve_lrec* Preserves Language

4.3.1 $Lang\ R\ A \subseteq Lang\ (solve\_lrec\ B\ B'\ R)\ A$

If there exists a derivation from u to v then there exists one which does not use productions of the form $A \to A$.

lemma *rm_self_loops_derivels*: **assumes** $P \vdash u \Rightarrow l(n)\ v$ **shows** *rm_self_loops* $P \vdash u \Rightarrow l{*}\ v$
proof −
have $rm\_self\_loops\ P = \{p{\in}P.\ \neg(\exists A.\ p = (A,[Nt\ A]))\}$ **unfolding** *rm_self_loops_def* **by** *auto*
with *no_self_loops_derivels*[*of n P u v*] *assms* **show** *?thesis* **by** *simp*
qed

Restricted to productions with one lhs (A), and no $A \to A$ productions if there is a derivation from u to A # v then u must start with Nt A.

lemma *lrec_lemma1*:
assumes $S = \{x.\ (\exists v.\ x = (A,\ v) \wedge x \in R)\}$ *rm_self_loops* $S \vdash u \Rightarrow l(n)\ Nt\ A\#v$
shows $\exists u'.\ u = Nt\ A\ \#\ u'$
proof (*rule ccontr*)
assume *neg*: $\nexists u'.\ u = Nt\ A\ \#\ u'$
show *False*
proof (*cases* $u = []$)
case *True*
then show *?thesis* **using** *assms* **by** *simp*
next
case *False*
then show *?thesis*
proof (*cases* $\exists t.\ hd\ u = Tm\ t$)
case *True*
then show *?thesis* **using** *assms neg*
by (*metis* (*no_types*, *lifting*) *False deriveln_Tm_Cons hd_Cons_tl list.inject*)
next
case *False*
then have $\exists B\ u'.\ u = Nt\ B\ \#\ u' \wedge B \neq A$ **using** *assms neg*
by (*metis deriveln_from_empty list.sel*(*1*) *neq_Nil_conv sym.exhaust*)
then obtain $B\ u'$ **where** *B_not_A*: $u = Nt\ B\ \#\ u' \wedge B \neq A$ **by** *blast*
then have $\exists w.\ (B,\ w) \in rm\_self\_loops\ S$ **using** *assms neg*
by (*metis* (*no_types*, *lifting*) *derivels_Nt_Cons relpowp_imp_rtranclp*)
then obtain w **where** *elem*: $(B,\ w) \in rm\_self\_loops\ S$ **by** *blast*
have $(B,\ w) \notin rm\_self\_loops\ S$ **using** *B_not_A assms* **by** (*auto simp add*: *rm_self_loops_def*)

    then show *?thesis* **using** *elem* **by** *simp*
  qed
 qed
qed

Restricted to productions with one lhs (A), and no $A \to A$ productions if there is a derivation from u to A # v then u must start with Nt A and there exists a prefix of A # v s.t. a left-derivation from [*Nt A*] to that prefix exists.

lemma *lrec_lemma2*:
 assumes $S = \{x.\ (\exists v.\ x = (A,\ v) \wedge x \in R)\}$ *Eps_free R*
 shows *rm_self_loops S* ⊢ *u* ⇒*l*(*n*) *Nt A*#*v* ⟹
  ∃ *u′ v′*. *u* = *Nt A* # *u′* ∧ *v* = *v′* @ *u′* ∧ (*rm_self_loops S*) ⊢ [*Nt A*] ⇒*l*(*n*) *Nt A* # *v′*
proof (*induction n arbitrary*: *u*)
 case *0*
 then show *?case* **by** *simp*
next
 case (*Suc n*)
 have ∃ *u′*. *u* = *Nt A* # *u′* **using** *lrec_lemma1*[*of S*] *Suc assms* **by** *auto*
 then obtain *u′* **where** *u′_prop*: *u* = *Nt A* # *u′* **by** *blast*
 then have ∃ *w*. (*A*,*w*) ∈ (*rm_self_loops S*) ∧ (*rm_self_loops S*) ⊢ *w* @ *u′* ⇒*l*(*n*) *Nt A* # *v*
  using *Suc* **by** (*auto simp add*: *deriveln_Nt_Cons split*: *prod.split*)
 then obtain *w* **where** *w_prop*:
  (*A*,*w*) ∈ (*rm_self_loops S*) ∧ (*rm_self_loops S*) ⊢ *w* @ *u′* ⇒*l*(*n*) *Nt A* # *v*
  by *blast*
 then have ∃ *u″ v″*. *w* @ *u′* = *Nt A* # *u″* ∧ *v* = *v″* @ *u″* ∧
  (*rm_self_loops S*) ⊢ [*Nt A*] ⇒*l*(*n*) *Nt A* # *v″*
  using *Suc.IH Suc* **by** *auto*
 then obtain *u″ v″* **where** *u″_prop*: *w* @ *u′* = *Nt A* # *u″* ∧ *v* = *v″* @ *u″* **and**
  ln_derive: (*rm_self_loops S*) ⊢ [*Nt A*] ⇒*l*(*n*) *Nt A* # *v″*
  by *blast*
 have *w* ≠ [] ∧ *w* ≠ [*Nt A*]
  using *Suc w_prop assms* **by** (*auto simp add*: *Eps_free_Nil rm_self_loops_def split*: *prod.splits*)
 then have ∃ *u1*. *u1* ≠ [] ∧ *w* = *Nt A* # *u1* ∧ *u″* = *u1* @ *u′*
  using *u″_prop* **by** (*metis Cons_eq_append_conv*)
 then obtain *u1* **where** *u1_prop*: *u1* ≠ [] ∧ *w* = *Nt A* # *u1* ∧ *u″* = *u1* @ *u′* **by** *blast*
 then have *1*: *u* = *Nt A* # *u′* ∧ *v* = (*v″* @ *u1*) @ *u′* **using** *u′_prop u″_prop* **by** *auto*

 have *2*: (*rm_self_loops S*) ⊢ [*Nt A*] @ *u1* ⇒*l*(*n*) *Nt A* # *v″* @ *u1*
  using *ln_derive deriveln_append*
  by *fastforce*
 have (*rm_self_loops S*) ⊢ [*Nt A*] ⇒*l* [*Nt A*] @ *u1*
  using *w_prop u″_prop u1_prop*
  by (*simp add*: *derivel_Nt_Cons*)

then have (*rm_self_loops S*) ⊢ [*Nt A*] ⇒*l*(*Suc n*) *Nt A* # *v''* @ *u1*
  using *ln_derive*
  by (*meson 2 relpowp_Suc_I2*)
then show *?case* **using** *1* **by** *blast*
qed

Restricted to productions with one lhs (A), and no $A \rightarrow A$ productions if there is a left-derivation from [*Nt A*] to A # u then there exists a derivation from [*Nt A'*] to u@[*Nt A*] and if $u \neq$ [] also to u in *solve_lrec A A' R*.

lemma *lrec_lemma3*:
  assumes *S* = {*x*. (∃ *v*. *x* = (*A*, *v*) ∧ *x* ∈ *R*)} *Eps_free R*
  shows *rm_self_loops S* ⊢ [*Nt A*] ⇒*l*(*n*) *Nt A* # *u*
  ⟹ *solve_lrec A A' S* ⊢ [*Nt A'*] ⇒(*n*) *u* @ [*Nt A'*] ∧
    (*u* ≠ [] ⟶ *solve_lrec A A' S* ⊢ [*Nt A'*] ⇒(*n*) *u*)
proof(*induction n arbitrary*: *u*)
  case *0*
  then show *?case* **by** (*simp*)
next
  case (*Suc n*)
  then have ∃ *w*. (*A*,*w*) ∈ *rm_self_loops S* ∧ *rm_self_loops S* ⊢ *w* ⇒*l*(*n*) *Nt A* # *u*
    by (*auto simp add*: *deriveln_Nt_Cons split*: *prod.splits*)
  then obtain *w* **where** *w_prop1*: (*A*,*w*) ∈ (*rm_self_loops S*) ∧ (*rm_self_loops S*) ⊢ *w* ⇒*l*(*n*) *Nt A*#*u*
    by *blast*
  then have ∃ *w' u'*. *w* = *Nt A* # *w'* ∧ *u* = *u'* @ *w'* ∧ (*rm_self_loops S*) ⊢ [*Nt A*] ⇒*l*(*n*) *Nt A* # *u'*
    using *lrec_lemma2*[*of S*] *Suc assms* **by** *auto*
  then obtain *w' u'* **where** *w_prop2*: *w* = *Nt A* # *w'* ∧ *u* = *u'* @ *w'* **and**
    ln_derive: *rm_self_loops S* ⊢ [*Nt A*] ⇒*l*(*n*) *Nt A* # *u'* **by** *blast*
  then have *w'* ≠ [] **using** *w_prop1 Suc* **by** (*auto simp add*: *rm_self_loops_def*)
  have (*A*, *w*) ∈ *S* **using** *Suc.prems*(*1*) *w_prop1* **by** (*auto simp add*: *rm_self_loops_def*)
  then have *prod_in_solve_lrec*: (*A'*, *w'* @ [*Nt A'*]) ∈ *solve_lrec A A' S*
    using *w_prop2* ‹*w'* ≠ []› **unfolding** *solve_lrec_defs* **by** (*auto*)

  have *1*: *solve_lrec A A' S* ⊢ [*Nt A'*] ⇒(*n*) *u'* @ [*Nt A'*] **using** *Suc.IH Suc ln_derive* **by** *auto*
  then have *2*: *solve_lrec A A' S* ⊢ [*Nt A'*] ⇒(*Suc n*) *u'* @ *w'* @ [*Nt A'*]
    using *prod_in_solve_lrec* **by** (*simp add*: *derive_prepend derive_singleton relpowp_Suc_I*)

  have (*A'*, *w'*) ∈ *solve_lrec A A' S* **using** *w_prop2* ‹*w'* ≠ []› ‹(*A*, *w*) ∈ *S*›
    unfolding *solve_lrec_defs* **by** (*auto*)
  then have *solve_lrec A A' S* ⊢ [*Nt A'*] ⇒(*Suc n*) *u'* @ *w'*
    using *1* **by** (*simp add*: *derive_prepend derive_singleton relpowp_Suc_I*)
  then show *?case* **using** *w_prop2 2* **by** *simp*
qed

A left derivation from p ($hd\ p = Nt\ A$) to q ($hd\ q \neq Nt\ A$) can be split into a left-recursive part, only using left-recursive productions $A \rightarrow A \# w$, one left derivation step consuming Nt A using some rule $A \rightarrow B \# v$ where $B \neq Nt\ A$ and a left-derivation comprising the rest of the derivation.

lemma *lrec_decomp*:
  assumes $S = \{x.\ (\exists v.\ x = (A, v) \wedge x \in R)\}$ *Eps_free R*
  shows $[\![\ hd\ p = Nt\ A;\ hd\ q \neq Nt\ A;\ R \vdash p \Rightarrow l(n)\ q\ ]\!]$
  $\Longrightarrow \exists u\ w\ m\ k.\ S \vdash p \Rightarrow l(m)\ Nt\ A \# u \wedge S \vdash Nt\ A \# u \Rightarrow l\ w \wedge hd\ w \neq Nt\ A \wedge$
      $R \vdash w \Rightarrow l(k)\ q \wedge n = m + k + 1$
proof (*induction n arbitrary*: *p*)
  case *0*
  then have *pq_not_Nil*: $p \neq [] \wedge q \neq []$ **using** *Eps_free_derives_Nil assms*
    by *simp*

  have $p = q$ **using** *0* **by** *auto*
  then show *?case* **using** *pq_not_Nil 0* **by** *auto*
next
  case (*Suc n*)
  then have *pq_not_Nil*: $p \neq [] \wedge q \neq []$
    using *Eps_free_deriveln_Nil assms* **by** *fastforce*

  have *ex_p'*: $\exists p'.\ p = Nt\ A \# p'$ **using** *pq_not_Nil Suc*
    by (*metis hd_Cons_tl*)
  then obtain p' **where** *P*: $p = Nt\ A \# p'$ **by** *blast*
  have $\nexists q'.\ q = Nt\ A \# q'$ **using** *pq_not_Nil Suc*
    by *fastforce*

  then have $\exists w.\ (A,w) \in R \wedge R \vdash w\ @\ p' \Rightarrow l(n)\ q$ **using** *Suc P* **by** (*auto simp add*: *deriveln_Nt_Cons*)
  then obtain *w* **where** *w_prop*: $(A,w) \in R \wedge R \vdash w\ @\ p' \Rightarrow l(n)\ q$ **by** *blast*
  then have *prod_in_S*: $(A, w) \in S$ **using** *Suc assms* **by** *auto*

  show *?case*
  proof (*cases* $\exists w'.\ w = Nt\ A \# w'$)
    case *True*
    then obtain w' **where** $w = Nt\ A \# w'$ **by** *blast*
    then have $\exists u\ w''\ m\ k.\ S \vdash w\ @\ p' \Rightarrow l(m)\ Nt\ A \# u \wedge S \vdash Nt\ A \# u \Rightarrow l\ w'' \wedge$
      $hd\ w'' \neq Nt\ A \wedge R \vdash w'' \Rightarrow l(k)\ q \wedge n = m + k + 1$
      using *Suc.IH Suc.prems w_prop* **by** *auto*
    then obtain $u\ w''\ m\ k$ **where** *propo*: $S \vdash w\ @\ p' \Rightarrow l(m)\ Nt\ A \# u \wedge S \vdash Nt\ A \# u \Rightarrow l\ w'' \wedge$
      $hd\ w'' \neq Nt\ A \wedge R \vdash w'' \Rightarrow l(k)\ q \wedge n = m + k + 1$
      by *blast*
    then have $S \vdash Nt\ A \# p' \Rightarrow l(Suc\ m)\ Nt\ A \# u$
      using *prod_in_S P* **by** (*meson derivel_Nt_Cons relpowp_Suc_I2*)

    then have $S \vdash p \Rightarrow l(Suc\ m)\ Nt\ A \# u \wedge S \vdash Nt\ A \# u \Rightarrow l\ w'' \wedge$
      $hd\ w'' \neq Nt\ A \wedge R \vdash w'' \Rightarrow l(k)\ q \wedge Suc\ n = Suc\ m + k + 1$
      using *P propo* **by** *auto*

```
    then show ?thesis by blast
  next
    case False
    then have w ≠ [] ∧ hd w ≠ Nt A using Suc w_prop assms
      by (metis Eps_free_Nil list.collapse)
    then have S ⊢ p ⇒l(0) Nt A # p' ∧ S ⊢ Nt A # p' ⇒l w @ p' ∧ hd (w @
p') ≠ Nt A ∧
         R ⊢ w @ p' ⇒l(n) q ∧ Suc n = 0 + n + 1
        using P w_prop prod_in_S by (auto simp add: derivel_Nt_Cons)
    then show ?thesis by blast
  qed
qed
```

Every derivation resulting in a word has a derivation in *solve_lrec B B' R*.

```
lemma tm_derive_impl_solve_lrec_derive:
  assumes Eps_free R B ≠ B' B' ∉ Nts R
  shows ⟦ p ≠ []; R ⊢ p ⇒(n) map Tm q⟧ ⟹ solve_lrec B B' R ⊢ p ⇒* map Tm
q
proof (induction n arbitrary: p q rule: nat_less_induct)
  case (1 n)
  then show ?case
  proof (cases solve_lrec B B' R = R − {(B, [Nt B])})
    case True
   have 2: rm_self_loops R ⊆ R − {(B, [Nt B])} by (auto simp add: rm_self_loops_def)
    have rm_self_loops R ⊢ p ⇒* map Tm q
    using rm_self_loops_derivels 1.prems(2) deriveln_iff_deriven derivels_imp_derives

      by blast
    then show ?thesis
      using 2 by (simp add: True derives_mono)
  next
    case solve_lrec_not_R: False
    then show ?thesis
    proof (cases nts_syms p = {})
      case True
     then obtain pt where p = map Tm pt using nts_syms_empty_iff by blast
      then have map Tm q = p
        using deriven_from_TmsD 1.prems(2) by blast
      then show ?thesis by simp
    next
      case False
    then have ∃ C pt p2. p = map Tm pt @ Nt C # p2 using non_word_has_first_Nt[of
p] by auto
      then obtain C pt p2 where P: p = map Tm pt @ Nt C # p2 by blast
      then have R ⊢ map Tm pt @ Nt C # p2 ⇒l(n) map Tm q
        using 1.prems by (simp add: deriveln_iff_deriven)
     then have ∃ q2. map Tm q = map Tm pt @ q2 ∧ R ⊢ Nt C # p2 ⇒l(n) q2
      by (simp add: deriveln_map_Tm_append[of n R pt Nt C # p2 map Tm q])
     then obtain q2 where P1: map Tm q = map Tm pt @ q2 ∧ R ⊢ Nt C #
```

```
p2 ⇒l(n) q2 by blast
      then have n ≠ 0
        by (metis False P nts_syms_map_Tm relpowp_0_E)
      then have ∃ m. n = Suc m
        by (meson old.nat.exhaust)
      then obtain m where n_Suc: n = Suc m by blast
      have ∃ q2t. q2 = map Tm q2t
        by (metis P1 append_eq_map_conv)
      then obtain q2t where q2_tms: q2 = map Tm q2t by blast
      then show ?thesis
      proof (cases C = B)
        case True
        then have n_derive: R ⊢ Nt B # p2 ⇒(n) q2 using P1
          by (simp add: deriveln_imp_deriven)
        have ∄ v2. q2 = Nt B #v2 ∧ R ⊢ p2 ⇒(n) v2 using q2_tms by auto
        then have ∃ n1 n2 w v1 v2. n = Suc (n1 + n2) ∧ q2 = v1 @ v2 ∧
             (B,w) ∈ R ∧ R ⊢ w ⇒(n1) v1 ∧ R ⊢ p2 ⇒(n2) v2 using n_derive
deriven_Cons_decomp
          by (smt (verit) sym.inject(1))
       then obtain n1 n2 w v1 v2 where decomp: n = Suc (n1 + n2) ∧ q2 = v1
@ v2 ∧
           (B,w) ∈ R ∧ R ⊢ w ⇒(n1) v1 ∧ R ⊢ p2 ⇒(n2) v2 by blast
        then have derive_from_singleton: R ⊢ [Nt B] ⇒(Suc n1) v1
          using deriven_Suc_decomp_left by force

        have v1 ≠ []
          using assms(1) Eps_free_deriven_Nil derive_from_singleton by auto
        then have ∃ v1t. v1 = map Tm v1t
          using decomp append_eq_map_conv q2_tms by blast
        then obtain v1t where v1_tms: v1 = map Tm v1t by blast
        then have v1_hd: hd v1 ≠ Nt B
          by (metis Nil_is_map_conv ‹v1 ≠ []› hd_map sym.distinct(1))

        have deriveln_from_singleton: R ⊢ [Nt B] ⇒l(Suc n1) v1 using v1_tms
derive_from_singleton
          by (simp add: deriveln_iff_deriven)
```

This is the interesting bit where we use other lemmas to prove that we can replace a specific part of the derivation which is a left-recursion by a right-recursion in the new productions.

```
        let ?S = {x. (∃ v. x = (B, v) ∧ x ∈ R)}
        have ∃ u w m k. ?S ⊢ [Nt B] ⇒l(m) Nt B # u ∧ ?S ⊢ Nt B # u ⇒l w ∧
          hd w ≠ Nt B ∧ R ⊢ w ⇒l(k) v1 ∧ Suc n1 = m + k + 1
          using deriveln_from_singleton v1_hd assms lrec_decomp[of ?S B R [Nt
B] v1 Suc n1] by auto
        then obtain u w2 m2 k where l_decomp: ?S ⊢ [Nt B] ⇒l(m2) Nt B # u
∧ ?S ⊢ Nt B # u ⇒l w2
           ∧ hd w2 ≠ Nt B ∧ R ⊢ w2 ⇒l(k) v1 ∧ Suc n1 = m2 + k + 1
          by blast
```

```
    then have ∃ w2'. (B,w2') ∈ ?S ∧ w2 = w2' @ u by (simp add: derivel_Nt_Cons)
    then obtain w2' where w2'_prod: (B,w2') ∈ ?S ∧ w2 = w2' @ u by blast
    then have w2'_props: w2' ≠ [] ∧ hd w2' ≠ Nt B
      by (metis (mono_tags, lifting) assms(1) Eps_free_Nil l_decomp
          hd_append mem_Collect_eq)

    have solve_lrec_subset: solve_lrec B B' ?S ⊆ solve_lrec B B' R
      unfolding solve_lrec_defs by (auto)

    have solve_lrec B B' ?S ⊢ [Nt B] ⇒* w2
      proof(cases u = [])
        case True
        have (B, w2') ∈ solve_lrec B B' ?S
          using w2'_props w2'_prod unfolding solve_lrec_defs by (auto)
        then show ?thesis
          by (simp add: True bu_prod derives_if_bu w2'_prod)
      next
        case False
        have solved_prod: (B, w2' @ [Nt B']) ∈ solve_lrec B B' ?S
        using w2'_props w2'_prod solve_lrec_not_R unfolding solve_lrec_defs by (auto)
        have rm_self_loops ?S ⊢ [Nt B] ⇒l* Nt B # u
          using l_decomp rm_self_loops_derivels by auto
        then have ∃ ln. rm_self_loops ?S ⊢ [Nt B] ⇒l(ln) Nt B # u
          by (simp add: rtranclp_power)
        then obtain ln where rm_self_loops ?S ⊢ [Nt B] ⇒l(ln) Nt B # u by blast
        then have (solve_lrec B B' ?S) ⊢ [Nt B'] ⇒(ln) u
          using lrec_lemma3[of ?S B R ln u] assms False by auto
        then have rrec_derive: (solve_lrec B B' ?S) ⊢ w2' @ [Nt B'] ⇒(ln) w2' @ u
          by (simp add: deriven_prepend)
        have (solve_lrec B B' ?S) ⊢ [Nt B] ⇒ w2' @ [Nt B']
          using solved_prod by (simp add: derive_singleton)
        then have (solve_lrec B B' ?S) ⊢ [Nt B] ⇒* w2' @ u
            using rrec_derive by (simp add: converse_rtranclp_into_rtranclp relpowp_imp_rtranclp)
        then show ?thesis using w2'_prod by auto
      qed
      then have 2: solve_lrec B B' R ⊢ [Nt B] ⇒* w2
        using solve_lrec_subset by (simp add: derives_mono)
```

From here on all the smaller derivations are concatenated after applying the IH.

```
     have fact2: R ⊢ w2 ⇒l(k) v1 ∧ Suc n1 = m2 + k + 1 using l_decomp by auto
    then have k < n
      using decomp by linarith
```

```
        then have 3: solve_lrec B B′ R ⊢ w2 ⇒* v1 using 1.IH v1_tms fact2
        by (metis deriveln_iff_deriven derives_from_empty relpowp_imp_rtranclp)

        have 4: solve_lrec B B′ R ⊢ [Nt B] ⇒* v1 using 2 3
          by auto

       have ∃ v2t. v2 = map Tm v2t using decomp append_eq_map_conv q2_tms
by blast
        then obtain v2t where v2_tms: v2 = map Tm v2t by blast
        have n2 < n using decomp by auto
        then have 5: solve_lrec B B′ R ⊢ p2 ⇒* v2 using 1.IH decomp v2_tms
          by (metis derives_from_empty relpowp_imp_rtranclp)

        have solve_lrec B B′ R ⊢ Nt B # p2 ⇒* q2 using 4 5 decomp
          by (metis append_Cons append_Nil derives_append_decomp)
        then show ?thesis
          by (simp add: P P1 True derives_prepend)
      next
        case C_not_B: False
        then have ∃ w. (C, w) ∈ R ∧ R ⊢ w @ p2 ⇒l(m) q2
          by (metis P1 derivel_Nt_Cons relpowp_Suc_D2 n_Suc)
        then obtain w where P2: (C, w) ∈ R ∧ R ⊢ w @ p2 ⇒l(m) q2 by blast
        then have rule_in_solve_lrec: (C, w) ∈ (solve_lrec B B′ R)
          using C_not_B by (auto simp add: solve_lrec_def rm_lrec_def)
       have derivem: R ⊢ w @ p2 ⇒(m) q2 using q2_tms P2 by (auto simp add:
deriveln_iff_deriven)
        have w @ p2 ≠ []
          using assms(1) Eps_free_Nil P2 by fastforce
       then have (solve_lrec B B′ R) ⊢ w @ p2 ⇒* q2 using 1.IH q2_tms n_Suc
derivem
          by auto
        then have (solve_lrec B B′ R) ⊢ Nt C # p2 ⇒* q2
          using rule_in_solve_lrec by (auto simp add: derives_Cons_rule)
        then show ?thesis
          by (simp add: P P1 derives_prepend)
      qed
    qed
  qed
qed

corollary Lang_incl_Lang_solve_lrec:
  ⟦ Eps_free R; B ≠ B′; B′ ∉ Nts R⟧ ⟹ Lang R A ⊆ Lang (solve_lrec B B′ R)
A
by(auto simp: Lang_def intro: tm_derive_impl_solve_lrec_derive dest: rtranclp_imp_relpowp)
```

4.3.2 Lang (solve_lrec B B′ R) A ⊆ Lang R A

Restricted to right-recursive productions of one Nt ($A' \rightarrow w$ @ $[Nt\ A']$) if there is a right-derivation from u to v @ $[Nt\ A']$ then u ends in Nt A'.

lemma *rrec_lemma1*:
assumes $S = \{x.\ \exists v.\ x = (A',\ v\ @\ [Nt\ A']) \wedge x \in solve\_lrec\ A\ A'\ R\}$ $S \vdash u \Rightarrow r(n)\ v\ @\ [Nt\ A']$
shows $\exists u'.\ u = u'\ @\ [Nt\ A']$
proof (*rule ccontr*)
assume *neg*: $\nexists u'.\ u = u'\ @\ [Nt\ A']$
show *False*
proof (*cases* $u = []$)
case *True*
then show *?thesis* **using** *assms derivern_imp_deriven* **by** *fastforce*
next
case *u_not_Nil*: *False*
then show *?thesis*
proof (*cases* $\exists t.\ last\ u = Tm\ t$)
case *True*
then show *?thesis* **using** *assms neg*
by (*metis* (*lifting*) *u_not_Nil append_butlast_last_id derivern_snoc_Tm last_snoc*)
next
case *False*
then have $\exists B\ u'.\ u = u'\ @\ [Nt\ B] \wedge B \neq A'$ **using** *assms neg u_not_Nil*
by (*metis append_butlast_last_id sym.exhaust*)
then obtain $B\ u'$ **where** *B_not_A'*: $u = u'\ @\ [Nt\ B] \wedge B \neq A'$ **by** *blast*
then have $\exists w.\ (B,\ w) \in S$ **using** *assms neg*
by (*metis* (*lifting*) *derivers_snoc_Nt relpowp_imp_rtranclp*)
then obtain w **where** *elem*: $(B,\ w) \in S$ **by** *blast*
have $(B,\ w) \notin S$ **using** *B_not_A' assms* **by** *auto*
then show *?thesis* **using** *elem* **by** *simp*
qed
qed
qed

solve_lrec does not add productions of the form $A' \rightarrow Nt\ A'$.

lemma *solve_lrec_no_self_loop*: $Eps\_free\ R \Longrightarrow A' \notin Nts\ R \Longrightarrow (A',\ [Nt\ A']) \notin solve\_lrec\ A\ A'\ R$
unfolding *solve_lrec_defs* **by** (*auto*)

Restricted to right-recursive productions of one Nt ($A' \rightarrow w\ @\ [Nt\ A']$) if there is a right-derivation from u to $v\ @\ [Nt\ A']$ then u ends in Nt A' and there exists a suffix of $v\ @\ [Nt\ A']$ s.t. there is a right-derivation from $[Nt\ A']$ to that suffix.

lemma *rrec_lemma2*:
assumes $S = \{x.\ (\exists v.\ x = (A',\ v\ @\ [Nt\ A']) \wedge x \in solve\_lrec\ A\ A'\ R)\}$ $Eps\_free\ R$ $A' \notin Nts\ R$
shows $S \vdash u \Rightarrow r(n)\ v\ @\ [Nt\ A']$
$\Longrightarrow \exists u'\ v'.\ u = u'\ @\ [Nt\ A'] \wedge v = u'\ @\ v' \wedge S \vdash [Nt\ A'] \Rightarrow r(n)\ v'\ @\ [Nt\ A']$
proof (*induction n arbitrary*: *u*)
case *0*
then show *?case* **by** *simp*

next
case (*Suc n*)
have $\exists u'.\ u = u'\ @\ [Nt\ A']$ **using** *rrec_lemma1*[*of S*] *Suc.prems assms* **by** *auto*
then obtain u' **where** *u'_prop*: $u = u'\ @\ [Nt\ A']$ **by** *blast*
then have $\exists w.\ (A',w) \in S \wedge S \vdash u'\ @\ w \Rightarrow r(n)\ v\ @\ [Nt\ A']$
using *Suc* **by** (*auto simp add*: *derivern_snoc_Nt*)
then obtain w **where** *w_prop*: $(A',w) \in S \wedge S \vdash u'\ @\ w \Rightarrow r(n)\ v\ @\ [Nt\ A']$ **by** *blast*
then have $\exists u''\ v''.\ u'\ @\ w = u''\ @\ [Nt\ A'] \wedge v = u''\ @\ v'' \wedge S \vdash [Nt\ A'] \Rightarrow r(n)\ v''\ @\ [Nt\ A']$
using *Suc.IH Suc* **by** *auto*
then obtain $u''\ v''$ **where** *u''_prop*: $u'\ @\ w = u''\ @\ [Nt\ A'] \wedge v = u''\ @\ v'' \wedge$
$S \vdash [Nt\ A'] \Rightarrow r(n)\ v''\ @\ [Nt\ A']$
by *blast*
have $w \neq [] \wedge w \neq [Nt\ A']$
using *Suc.IH assms w_prop solve_lrec_no_self_loop* **by** *fastforce*
then have $\exists u1.\ u1 \neq [] \wedge w = u1\ @\ [Nt\ A'] \wedge u'' = u'\ @\ u1$
using *u''_prop*
by (*metis* (*no_types*, *opaque_lifting*) *append.left_neutral append1_eq_conv append_assoc rev_exhaust*)
then obtain *u1* **where** *u1_prop*: $u1 \neq [] \wedge w = u1\ @\ [Nt\ A'] \wedge u'' = u'\ @\ u1$ **by** *blast*
then have *1*: $u = u'\ @\ [Nt\ A'] \wedge v = u'\ @\ (u1\ @\ v'')$ **using** *u'_prop u''_prop* **by** *auto*

have *2*: $S \vdash u1\ @\ [Nt\ A'] \Rightarrow r(n)\ u1\ @\ v''\ @\ [Nt\ A']$ **using** *u''_prop derivern_prepend*
by *fastforce*
have $S \vdash [Nt\ A'] \Rightarrow r\ u1\ @\ [Nt\ A']$ **using** *w_prop u''_prop u1_prop*
by (*simp add*: *deriver_singleton*)
then have $S \vdash [Nt\ A'] \Rightarrow r(Suc\ n)\ u1\ @\ v''\ @\ [Nt\ A']$ **using** *u''_prop*
by (*meson 2 relpowp_Suc_I2*)
then show *?case* **using** *1*
by *auto*
qed

Restricted to right-recursive productions of one Nt ($A' \rightarrow w\ @\ [Nt\ A']$) if there is a restricted right-derivation in *solve_lrec* from $[Nt\ A']$ to $u\ @\ [Nt\ A']$ then there exists a derivation in R from $[Nt\ A]$ to $A\ \#\ u$.

lemma *rrec_lemma3*:
assumes $S = \{x.\ (\exists v.\ x = (A',\ v\ @\ [Nt\ A']) \wedge x \in solve\_lrec\ A\ A'\ R)\}$ *Eps_free R*
$A' \notin Nts\ R\ A \neq A'$
shows $S \vdash [Nt\ A'] \Rightarrow r(n)\ u\ @\ [Nt\ A'] \Longrightarrow R \vdash [Nt\ A] \Rightarrow (n)\ Nt\ A\ \#\ u$
proof(*induction n arbitrary*: *u*)
case *0*
then show *?case* **by** (*simp*)
next
case (*Suc n*)

then have ∃ w. (A′,w) ∈ S ∧ S ⊢ w ⇒r(n) u @ [Nt A′]
 by (auto simp add: derivern_singleton split: prod.splits)
then obtain w **where** w_prop1: (A′,w) ∈ S ∧ S ⊢ w ⇒r(n) u @ [Nt A′] **by** blast
then have ∃ u′ v′. w = u′ @ [Nt A′] ∧ u = u′ @ v′ ∧ S ⊢ [Nt A′] ⇒r(n) v′ @ [Nt A′]
 using rrec_lemma2[of S] assms **by** auto
then obtain u′ v′ **where** u′v′_prop: w = u′ @ [Nt A′] ∧ u = u′ @ v′
 ∧ S ⊢ [Nt A′] ⇒r(n) v′ @ [Nt A′]
 by blast
then have 1: R ⊢ [Nt A] ⇒(n) Nt A # v′ **using** Suc.IH **by** auto

have (A′, u′ @ [Nt A′]) ∈ solve_lrec A A′ R ⟶ (A, Nt A # u′) ∈ R
 using assms **unfolding** solve_lrec_defs **by** (auto)
then have (A, Nt A # u′) ∈ R **using** u′v′_prop assms(1) w_prop1 **by** auto

then have R ⊢ [Nt A] ⇒ Nt A # u′
 by (simp add: derive_singleton)
then have R ⊢ [Nt A] @ v′ ⇒ Nt A # u′ @ v′
 by (metis Cons_eq_appendI derive_append)
then have R ⊢ [Nt A] ⇒(Suc n) Nt A # (u′ @ v′) **using** 1
 by (simp add: relpowp_Suc_I)
then show ?case **using** u′v′_prop **by** simp
qed

A right derivation from p@[Nt A′] to q (last q ≠ Nt A′) can be split into a right-recursive part, only using right-recursive productions with Nt A′, one right derivation step consuming Nt A′ using some rule A′ → as@[Nt B] where Nt B ≠ Nt A′ and a right-derivation comprising the rest of the derivation.

lemma rrec_decomp:
assumes S = {x. (∃ v. x = (A′, v @ [Nt A′]) ∧ x ∈ solve_lrec A A′ R)} Eps_free R
 A ≠ A′ A′ ∉ Nts R
shows ⟦A′ ∉ nts_syms p; last q ≠ Nt A′; solve_lrec A A′ R ⊢ p @ [Nt A′] ⇒r(n) q⟧
 ⟹ ∃ u w m k. S ⊢ p @ [Nt A′] ⇒r(m) u @ [Nt A′]
 ∧ solve_lrec A A′ R ⊢ u @ [Nt A′] ⇒r w ∧ A′ ∉ nts_syms w
 ∧ solve_lrec A A′ R ⊢ w ⇒r(k) q ∧ n = m + k + 1
proof (induction n arbitrary: p)
case 0
then have pq_not_Nil: p @ [Nt A′] ≠ [] ∧ q ≠ [] **using** Eps_free_derives_Nil
 by auto

have p = q **using** 0 **by** auto
then show ?case **using** pq_not_Nil 0 **by** auto
next
case (Suc n)
have pq_not_Nil: p @ [Nt A′] ≠ [] ∧ q ≠ []

```
  using assms Suc.prems Eps_free_deriven_Nil Eps_free_solve_lrec derivern_imp_deriven
    by (metis (no_types, lifting) snoc_eq_iff_butlast)

  have ∄ q'. q = q' @ [Nt A'] using pq_not_Nil Suc.prems
    by fastforce

  then have ∃ w. (A',w) ∈ (solve_lrec A A' R) ∧ (solve_lrec A A' R) ⊢ p @ w
⇒r(n) q
    using Suc.prems by (auto simp add: derivern_snoc_Nt)
  then obtain w where w_prop: (A',w) ∈ (solve_lrec A A' R) ∧ solve_lrec A A'
R ⊢ p @ w ⇒r(n) q
    by blast

  show ?case
  proof (cases (A', w) ∈ S)
    case True
    then have ∃ w'. w = w' @ [Nt A']
      by (simp add: assms(1))
    then obtain w' where w_decomp: w = w' @ [Nt A'] by blast
    then have A' ∉ nts_syms (p @ w') using assms Suc.prems True
      unfolding solve_lrec_defs by (auto split: if_splits)
    then have ∃ u w'' m k. S ⊢ p @ w ⇒r(m) u @ [Nt A'] ∧ solve_lrec A A' R ⊢
u @ [Nt A'] ⇒r w''
      ∧ A' ∉ nts_syms w'' ∧ solve_lrec A A' R ⊢ w'' ⇒r(k) q ∧ n = m + k + 1
     using Suc.IH Suc.prems w_prop w_decomp by (metis (lifting) append_assoc)
    then obtain u w'' m k where propo:
      S ⊢ p @ w ⇒r(m) u @ [Nt A'] ∧ solve_lrec A A' R ⊢ u @ [Nt A'] ⇒r w'' ∧
A' ∉ nts_syms w''
       ∧ solve_lrec A A' R ⊢ w'' ⇒r(k) q ∧ n = m + k + 1
      by blast
    then have S ⊢ p @ [Nt A'] ⇒r(Suc m) u @ [Nt A'] using True
      by (meson deriver_snoc_Nt relpowp_Suc_I2)

    then have S ⊢ p @ [Nt A'] ⇒r(Suc m) u @ [Nt A'] ∧ solve_lrec A A' R ⊢ u
@ [Nt A'] ⇒r w''
       ∧ A' ∉ nts_syms w'' ∧ solve_lrec A A' R ⊢ w'' ⇒r(k) q ∧ Suc n = Suc m
+ k + 1
      using propo by auto
    then show ?thesis by blast
  next
    case False
    then have last w ≠ Nt A' using assms
      by (metis (mono_tags, lifting) Eps_freeE_Cons Eps_free_solve_lrec
          append_butlast_last_id list.distinct(1) mem_Collect_eq w_prop)
    then have A' ∉ nts_syms w using assms w_prop
      unfolding solve_lrec_defs by (auto split: if_splits)
    then have w ≠ [] ∧ A' ∉ nts_syms w using assms w_prop False
      by (metis (mono_tags, lifting) Eps_free_Nil Eps_free_solve_lrec)
    then have S ⊢ p @ [Nt A'] ⇒r(0) p @ [Nt A'] ∧ solve_lrec A A' R ⊢ p @ [Nt
```

A′] ⇒r p @ w
∧ A′ ∉ nts_syms (p @ w) ∧ solve_lrec A A′ R ⊢ p @ w ⇒r(n) q ∧ Suc n = 0 + n + 1
using *w_prop Suc.prems* **by** (*auto simp add*: *deriver_snoc_Nt*)
then show *?thesis* **by** *blast*
qed
qed

Every word derived by *solve_lrec B B′ R* can be derived by *R*.

lemma *tm_solve_lrec_derive_impl_derive*:
assumes *Eps_free R B ≠ B′ B′ ∉ Nts R*
shows ⟦ *p ≠ []*; *B′ ∉ nts_syms p*; (*solve_lrec B B′ R*) ⊢ *p ⇒(n) map Tm q*⟧ ⟹ *R ⊢ p ⇒∗ map Tm q*
proof (*induction arbitrary*: *p q rule*: *nat_less_induct*)
case (*1 n*)
let *?R′* = (*solve_lrec B B′ R*)
show *?case*
proof (*cases nts_syms p* = {})
case *True*
then show *?thesis*
using *1.prems*(*3*) *deriven_from_TmsD derives_from_Tms_iff*
by (*metis nts_syms_empty_iff*)
next
case *False*
from *non_word_has_last_Nt*[*OF this*] **have** ∃ *C pt p2*. *p* = *p2* @ [*Nt C*] @ *map Tm pt* **by** *blast*
then obtain *C pt p2* **where** *p_decomp*: *p* = *p2* @ [*Nt C*] @ *map Tm pt* **by** *blast*
then have ∃ *pt′ At w k m*. *?R′* ⊢ *p2* ⇒(*k*) *map Tm pt′* ∧ *?R′* ⊢ *w* ⇒(*m*) *map Tm At* ∧ (*C*, *w*) ∈ *?R′*
∧ *q* = *pt′* @ *At* @ *pt* ∧ *n* = *Suc*(*k* + *m*)
using *1.prems word_decomp1*[*of n ?R′ p2 C pt q*] **by** *auto*
then obtain *pt′ At w k m*
where *P*: *?R′* ⊢ *p2* ⇒(*k*) *map Tm pt′* ∧ *?R′* ⊢ *w* ⇒(*m*) *map Tm At* ∧ (*C*, *w*) ∈ *?R′*
∧ *q* = *pt′* @ *At* @ *pt* ∧ *n* = *Suc*(*k* + *m*)
by *blast*
then have *pre1*: *m* < *n* **by** *auto*

have *B′* ∉ *nts_syms p2* ∧ *k* < *n* **using** *P 1.prems p_decomp* **by** *auto*
then have *p2_not_Nil_derive*: *p2* ≠ [] ⟶ *R* ⊢ *p2* ⇒∗ *map Tm pt′* **using** *1 P* **by** *blast*

have *p2* = [] ⟶ *map Tm pt′* = [] **using** *P*
by *auto*
then have *p2_derive*: *R* ⊢ *p2* ⇒∗ *map Tm pt′* **using** *p2_not_Nil_derive* **by** *auto*

have *R* ⊢ [*Nt C*] ⇒∗ *map Tm At*

proof(*cases C = B*)
case *C_is_B*: *True*
then show *?thesis*
proof (*cases last w = Nt B′*)
case *True*
let *?S = {x. (∃ v. x = (B′, v @ [Nt B′]) ∧ x ∈ solve_lrec B B′ R)}*

have *∃ w1. w = w1 @ [Nt B′]* **using** *True*
by (*metis assms(1) Eps_free_Nil Eps_free_solve_lrec P append_butlast_last_id*)
then obtain *w1* **where** *w_decomp*: *w = w1 @ [Nt B′]* **by** *blast*
then have *∃ w1′ b k1 m1. ?R′ ⊢ w1 ⇒(k1) w1′ ∧ ?R′ ⊢ [Nt B′] ⇒(m1) b ∧ map Tm At = w1′ @ b*
∧ m = k1 + m1
using *P deriven_append_decomp* **by** *blast*
then obtain *w1′ b k1 m1*
where *w_derive_decomp*: *?R′ ⊢ w1 ⇒(k1) w1′ ∧ ?R′ ⊢ [Nt B′] ⇒(m1) b*
∧ map Tm At = w1′ @ b ∧ m = k1 + m1
by *blast*
then have *∃ w1t bt. w1′ = map Tm w1t ∧ b = map Tm bt*
by (*meson map_eq_append_conv*)
then obtain *w1t bt* **where** *tms*: *w1′ = map Tm w1t ∧ b = map Tm bt* **by** *blast*

have *pre1*: *k1 < n ∧ m1 < n* **using** *w_derive_decomp P* **by** *auto*
have *pre2*: *w1 ≠ []* **using** *w_decomp C_is_B P assms* **by** (*auto simp add*: *solve_lrec_rule_simp3*)
have *Bw1_in_R*: *(B, w1) ∈ R*
using *w_decomp P C_is_B assms*
unfolding *solve_lrec_defs* **by** (*auto split*: *if_splits*)

then have *pre3*: *B′ ∉ nts_syms w1* **using** *assms* **by** (*auto simp add*: *Nts_def*)

have *R ⊢ w1 ⇒∗ map Tm w1t* **using** *pre1 pre2 pre3 w_derive_decomp 1.IH tms* **by** *blast*
then have *w1′_derive*: *R ⊢ [Nt B] ⇒∗ w1′* **using** *Bw1_in_R tms*
by (*simp add*: *derives_Cons_rule*)

have *last [Nt B′] = Nt B′ ∧ last (map Tm bt) ≠ Nt B′*
by (*metis assms(1) Eps_free_deriven_Nil Eps_free_solve_lrec last_ConsL last_map*
list.map_disc_iff not_Cons_self2 sym.distinct(1) tms w_derive_decomp)
then have *∃ u v m2 k2. ?S ⊢ [Nt B′] ⇒r(m2) u @ [Nt B′] ∧ ?R′ ⊢ u @ [Nt B′] ⇒r v*
∧ B′ ∉ nts_syms v ∧ ?R′ ⊢ v ⇒r(k2) map Tm bt ∧ m1 = m2 + k2 + 1
using *rrec_decomp[of ?S B′ B R [] map Tm bt m1] w_derive_decomp assms 1.prems tms*
by (*simp add*: *derivern_iff_deriven*)

then obtain *u v m2 k2*
where *rec_decomp*: *?S* ⊢ [*Nt B'*] ⇒*r*(*m2*) *u* @ [*Nt B'*] ∧ *?R'* ⊢ *u* @ [*Nt B'*] ⇒*r v*
∧ *B'* ∉ *nts_syms v* ∧ *?R'* ⊢ *v* ⇒*r*(*k2*) *map Tm bt* ∧ *m1* = *m2* + *k2* + *1*
by *blast*
then have *Bu_derive*: *R* ⊢ [*Nt B*] ⇒(*m2*) *Nt B* # *u*
using *assms rrec_lemma3* **by** *fastforce*

have ∃ *v'*. (*B'*, *v'*) ∈ *?R'* ∧ *v* = *u* @ *v'* **using** *rec_decomp*
by (*simp add*: *deriver_snoc_Nt*)
then obtain *v'* **where** *v_decomp*: (*B'*, *v'*) ∈ *?R'* ∧ *v* = *u* @ *v'* **by** *blast*
then have (*B*, *Nt B* # *v'*) ∈ *R*
using *assms rec_decomp* **unfolding** *solve_lrec_defs* **by** (*auto split*: *if_splits*)
then have *R* ⊢ [*Nt B*] ⇒ *Nt B* # *v'*
by (*simp add*: *derive_singleton*)
then have *R* ⊢ [*Nt B*] @ *v'* ⇒∗ *Nt B* # *u* @ *v'*
by (*metis Bu_derive append_Cons derives_append rtranclp_power*)
then have *Buv'_derive*: *R* ⊢ [*Nt B*] ⇒∗ *Nt B* # *u* @ *v'*
using ‹*R* ⊢ [*Nt B*] ⇒ *Nt B* # *v'*› **by** *force*

have *pre2*: *k2* < *n* **using** *rec_decomp pre1* **by** *auto*
have *v* ≠ [] **using** *rec_decomp*
by (*metis* (*lifting*) *assms*(*1*) *Eps_free_deriven_Nil Eps_free_solve_lrec tms*
deriven_from_TmsD derivern_imp_deriven list.simps(*8*) *not_Cons_self2 w_derive_decomp*)
then have *R* ⊢ *v* ⇒∗ *map Tm bt*
using *1.IH 1 pre2 rec_decomp*
by (*auto simp add*: *derivern_iff_deriven*)
then have *R* ⊢ [*Nt B*] ⇒∗ *Nt B* # *map Tm bt* **using** *Buv'_derive v_decomp*
by (*meson derives_Cons rtranclp_trans*)
then have *R* ⊢ [*Nt B*] ⇒∗ [*Nt B*] @ *map Tm bt* **by** *auto*
then have *R* ⊢ [*Nt B*] ⇒∗ *w1'* @ *map Tm bt* **using** *w1'_derive derives_append*
by (*metis rtranclp_trans*)
then show *?thesis* **using** *tms w_derive_decomp C_is_B* **by** *auto*
next
case *False*
have *pre2*: *w* ≠ [] **using** *P assms*(*1*)
by (*meson Eps_free_Nil Eps_free_solve_lrec*)
then have *2*: (*C*, *w*) ∈ *R*
using *P False 1.prems p_decomp C_is_B*
unfolding *solve_lrec_defs* **by** (*auto split*: *if_splits*)

then have *pre3*: *B'* ∉ *nts_syms w* **using** *P assms*(*3*) **by** (*auto simp add*: *Nts_def*)

have *R* ⊢ *w* ⇒∗ *map Tm At* **using** *1.IH assms pre1 pre2 pre3 P* **by** *blast*

then show *?thesis* **using** *2*
by (*meson bu_prod derives_bu_iff rtranclp_trans*)
qed
next
case *False*
then have *2*: $(C, w) \in R$
using *P 1.prems(2) p_decomp*
by (*auto simp add*: *solve_lrec_rule_simp1*)
then have *pre2*: $B' \notin$ *nts_syms w* **using** *P assms(3)* **by** (*auto simp add*: *Nts_def*)
have *pre3*: $w \neq []$ **using** *assms(1) 2* **by** (*auto simp add*: *Eps_free_def*)

have $R \vdash w \Rightarrow*$ *map Tm At* **using** *1.IH pre1 pre2 pre3 P* **by** *blast*
then show *?thesis* **using** *2*
by (*meson bu_prod derives_bu_iff rtranclp_trans*)
qed

then show *?thesis* **using** *p2_derive*
by (*metis P derives_append derives_append_decomp map_append p_decomp*)
qed
qed

corollary *Lang_solve_lrec_incl_Lang*:
assumes *Eps_free R* $B \neq B'$ $B' \notin$ *Nts R* $A \neq B'$
shows *Lang* (*solve_lrec B B' R*) $A \subseteq$ *Lang R A*
proof
fix *w*
assume $w \in$ *Lang* (*solve_lrec B B' R*) *A*
then have *solve_lrec B B' R* $\vdash$ [*Nt A*] $\Rightarrow*$ *map Tm w* **by** (*simp add*: *Lang_def*)
then have $\exists n$. *solve_lrec B B' R* $\vdash$ [*Nt A*] $\Rightarrow(n)$ *map Tm w*
by (*simp add*: *rtranclp_power*)
then obtain *n* **where** (*solve_lrec B B' R*) $\vdash$ [*Nt A*] $\Rightarrow(n)$ *map Tm w* **by** *blast*
then have $R \vdash$ [*Nt A*] $\Rightarrow*$ *map Tm w* **using** *tm_solve_lrec_derive_impl_derive*[*of R*] *assms* **by** *auto*
then show $w \in$ *Lang R A* **by** (*simp add*: *Lang_def*)
qed

corollary *solve_lrec_Lang*:
⟦ *Eps_free R*; $B \neq B'$; $B' \notin$ *Nts R*; $A \neq B'$⟧ $\Longrightarrow$ *Lang* (*solve_lrec B B' R*) *A* = *Lang R A*
using *Lang_solve_lrec_incl_Lang Lang_incl_Lang_solve_lrec* **by** *fastforce*

4.4 *expand_hd* Preserves Language

Every rhs of an *expand_hd R* production is derivable by *R*.

lemma *expand_hd_is_deriveable*: $(A, w) \in$ *expand_hd B As R* $\Longrightarrow R \vdash$ [*Nt A*] $\Rightarrow*$ *w*
proof (*induction B As R arbitrary*: *A w rule*: *expand_hd.induct*)
case (*1 B R*)

```
  then show ?case
    by (simp add: bu_prod derives_if_bu)
next
  case (2 B S Ss R)
  then show ?case
  proof (cases B = A)
    case True
    then have Aw_or_ACv: (A, w) ∈ expand_hd A Ss R ∨ (∃ C v. (A, Nt C #
v) ∈ expand_hd A Ss R)
      using 2 by (auto simp add: Let_def)
    then show ?thesis
    proof (cases (A, w) ∈ expand_hd A Ss R)
      case True
      then show ?thesis using 2 True by (auto simp add: Let_def)
    next
      case False
      then have ∃ v wv. w = v @ wv ∧ (A, Nt S#wv) ∈ expand_hd A Ss R ∧ (S,
v) ∈ expand_hd A Ss R
        using 2 True by (auto simp add: Let_def)
      then obtain v wv
        where P: w = v @ wv ∧ (A, Nt S # wv) ∈ expand_hd A Ss R ∧ (S, v) ∈
expand_hd A Ss R
        by blast
      then have tr: R ⊢ [Nt A] ⇒∗ [Nt S] @ wv using 2 True by simp
      have R ⊢ [Nt S] ⇒∗ v using 2 True P by simp
      then show ?thesis using P tr derives_append
        by (metis rtranclp_trans)
    qed
  next
    case False
    then show ?thesis using 2 by (auto simp add: Let_def)
  qed
qed
```

```
lemma expand_hd_incl1: Lang (expand_hd B As R) A ⊆ Lang R A
by (meson DersD DersI Lang_subset_if_Ders_subset derives_simul_rules expand_hd_is_deriveable subsetI)
```

This lemma expects a set of quadruples (A, $a1$, B, $a2$). Each quadruple encodes a specific Nt in a specific rule $A \to a1$ @ $Nt\ B$ # $a2$ (this encodes Nt B) which should be expanded, by replacing the Nt with every rule for that Nt and then removing the original rule. This expansion contains the original productions Language.

```
lemma exp_includes_Lang:
  assumes S_props: ∀ x ∈ S. ∃ A a1 B a2. x = (A, a1, B, a2) ∧ (A, a1 @ Nt B
# a2) ∈ R
  shows Lang R A
      ⊆ Lang (R − {x. ∃ A a1 B a2. x = (A, a1 @ Nt B # a2) ∧ (A, a1, B, a2)
```

```
∈ S }
          ∪ {x. ∃ A v a1 a2 B. x = (A,a1@v@a2) ∧ (A, a1, B, a2) ∈ S ∧ (B,v)
∈ R}) A
proof
  fix x
  assume x_Lang: x ∈ Lang R A
  let ?S' = {x. ∃ A a1 B a2. x = (A, a1 @ Nt B # a2) ∧ (A, a1, B, a2) ∈ S }
  let ?E = {x. ∃ A v a1 a2 B. x = (A,a1@v@a2) ∧ (A, a1, B, a2) ∈ S ∧ (B,v)
∈ R}
  let ?subst = R − ?S' ∪ ?E
  have S'_sub: ?S' ⊆ R using S_props by auto
  have (N, ts) ∈ ?S' ⟹ ∃ B. B ∈ nts_syms ts for N ts by fastforce
  then have terminal_prods_stay: (N, ts) ∈ R ⟹ nts_syms ts = {} ⟹ (N, ts)
∈ ?subst for N ts
    by auto

  have R ⊢ p ⇒(n) map Tm x ⟹ ?subst ⊢ p ⇒* map Tm x for p n
  proof (induction n arbitrary: p x rule: nat_less_induct)
    case (1 n)
    then show ?case
    proof (cases ∃ pt. p = map Tm pt)
      case True
      then obtain pt where p = map Tm pt by blast
     then show ?thesis using 1.prems deriven_from_TmsD derives_from_Tms_iff
by blast
    next
      case False
      then have ∃ uu V ww. p = uu @ Nt V # ww
        by (smt (verit, best) 1.prems deriven_Suc_decomp_left relpowp_E)
      then obtain uu V ww where p_eq: p = uu @ Nt V # ww by blast
      then have ¬ R ⊢ p ⇒(0) map Tm x
        using False by auto
      then have ∃ m. n = Suc m
        using 1.prems old.nat.exhaust by blast
      then obtain m where n_Suc: n = Suc m by blast
      then have ∃ v. (V, v) ∈ R ∧ R ⊢ uu @ v @ ww ⇒(m) map Tm x
        using 1 p_eq by (auto simp add: deriven_start_sent)
     then obtain v where start_deriven: (V, v) ∈ R ∧ R ⊢ uu @ v @ ww ⇒(m)
map Tm x by blast
      then show ?thesis
      proof (cases (V, v) ∈ ?S')
        case True
       then have ∃ a1 B a2. v = a1 @ Nt B # a2 ∧ (V, a1, B, a2) ∈ S by blast
       then obtain a1 B a2 where v_eq: v = a1 @ Nt B # a2 ∧ (V, a1, B, a2)
∈ S by blast
        then have m_deriven: R ⊢ (uu @ a1) @ Nt B # (a2 @ ww) ⇒(m) map
Tm x
          using start_deriven by auto
        then have ¬ R ⊢ (uu @ a1) @ Nt B # (a2 @ ww) ⇒(0) map Tm x
```

```
            by (metis (mono_tags, lifting) append.left_neutral append_Cons derive.intros insertI1
             not_derive_from_Tms relpowp.simps(1))
        then have ∃k. m = Suc k
          using m_deriven 1.prems old.nat.exhaust by blast
        then obtain k where m_Suc: m = Suc k by blast
        then have ∃b. (B, b) ∈ R ∧ R ⊢ (uu @ a1) @ b @ (a2 @ ww) ⇒(k) map Tm x
          using m_deriven deriven_start_sent[where ?u = uu@a1 and ?w = a2 @ ww]
          by (auto simp add: m_Suc)
        then obtain b
          where second_deriven: (B, b) ∈ R ∧ R ⊢ (uu @ a1) @ b @ (a2 @ ww) ⇒(k) map Tm x
          by blast
        then have expd_rule_subst: (V, a1 @ b @ a2) ∈ ?subst using v_eq by auto
        have k < n using n_Suc m_Suc by auto
        then have subst_derives: ?subst ⊢ uu @ a1 @ b @ a2 @ ww ⇒* map Tm x
          using 1 second_deriven by (auto)
        have ?subst ⊢ [Nt V] ⇒* a1 @ b @ a2 using expd_rule_subst
          by (meson derive_singleton r_into_rtranclp)
        then have ?subst ⊢ [Nt V] @ ww ⇒* a1 @ b @ a2 @ ww
          using derives_append[of ?subst [Nt V] a1 @ b @ a2]
          by simp
        then have ?subst ⊢ Nt V # ww ⇒* a1 @ b @ a2 @ ww
          by simp
        then have ?subst ⊢ uu @ Nt V # ww ⇒* uu @ a1 @ b @ a2 @ ww
          using derives_prepend[of ?subst [Nt V] @ ww]
          by simp
        then show ?thesis using subst_derives by (auto simp add: p_eq v_eq)
      next
        case False
        then have Vv_subst: (V,v) ∈ ?subst using S_props start_deriven by auto
        then have ?subst ⊢ uu @ v @ ww ⇒* map Tm x using 1 start_deriven n_Suc by auto
        then show ?thesis using Vv_subst derives_append_decomp
          by (metis (no_types, lifting) derives_Cons_rule p_eq)
      qed
    qed
  qed

  then have R ⊢ p ⇒* map Tm x ⟹ ?subst ⊢ p ⇒* map Tm x for p
    by (meson rtranclp_power)

  then show x ∈ Lang ?subst A using x_Lang by (auto simp add: Lang_def)
qed
```

lemma *expand_hd_incl2*: *Lang* (*expand_hd B As R*) *A* ⊇ *Lang R A*
proof (*induction B As R rule*: *expand_hd.induct*)
  case (*1 A R*)
  then show *?case* **by** *simp*
next
  case (*2 C H Ss R*)
  let *?R'* = *expand_hd C Ss R*
  let *?X* = {(*Al*,*Bw*) ∈ *?R'*. *Al*=*C* ∧ (∃ *w*. *Bw* = *Nt H* # *w*)}
  let *?Y* = {(*C*,*v*@*w*) |*v w*. ∃ *B*. (*C*,*Nt B* # *w*) ∈ *?X* ∧ (*B*,*v*) ∈ *?R'*}
  have *expand_hd C* (*H* # *Ss*) *R* = *?R'* − *?X* ∪ *?Y* **by** (*simp add*: *Let_def*)

  let *?S* = {*x*. ∃ *A w*. *x* = (*A*, [], *H*, *w*) ∧ (*A*, *Nt H* # *w*) ∈ *?X*}
  let *?S'* = {*x*. ∃ *A a1 B a2*. *x* = (*A*, *a1* @ *Nt B* # *a2*) ∧ (*A*, *a1*, *B*, *a2*) ∈ *?S*}
  let *?E* = {*x*. ∃ *A v a1 a2 B*. *x* = (*A*,*a1*@*v*@*a2*) ∧ (*A*, *a1*, *B*, *a2*) ∈ *?S* ∧ (*B*,*v*) ∈ *?R'*}

  have *S'_eq_X*: *?S'* = *?X* **by** *fastforce*
  have *E_eq_Y*: *?E* = *?Y* **by** *fastforce*

  have ∀ *x* ∈ *?S*. ∃ *A a1 B a2*. *x* = (*A*, *a1*, *B*, *a2*) ∧ (*A*, *a1* @ *Nt B* # *a2*) ∈ *?R'* **by** *fastforce*
  then have *Lang_sub*: *Lang ?R' A* ⊆ *Lang* (*?R'* − *?S'* ∪ *?E*) *A*
    using *exp_includes_Lang*[*of ?S*] **by** *auto*

  have *Lang R A* ⊆ *Lang ?R' A* **using** *2* **by** *simp*
  also have ... ⊆ *Lang* (*?R'* − *?S'* ∪ *?E*) *A* **using** *Lang_sub* **by** *simp*
  also have ... ⊆ *Lang* (*?R'* − *?X* ∪ *?Y*) *A* **using** *S'_eq_X E_eq_Y* **by** *simp*
  finally show *?case* **by** (*simp add*: *Let_def*)
qed

theorem *expand_hd_Lang*: *Lang* (*expand_hd B As R*) *A* = *Lang R A*
  using *expand_hd_incl1*[*of B As R A*] *expand_hd_incl2*[*of R A B As*] **by** *auto*

4.5 *solve_tri* Preserves Language

lemma *solve_tri_Lang*:
  ⟦ *Eps_free R*; *length As* ≤ *length As'*; *distinct*(*As* @ *As'*); *Nts R* ∩ *set As'* = {}; *A* ∉ *set As*⟧
    ⟹ *Lang* (*solve_tri As As' R*) *A* = *Lang R A*
proof (*induction As As' R rule*: *solve_tri.induct*)
  case (*1 uu R*)
  then show *?case* **by** *simp*
next
  case (*2 Aa As A' As' R*)
  then have *e_free1*: *Eps_free* (*expand_hd Aa As* (*solve_tri As As' R*))
    by (*simp add*: *Eps_free_expand_hd Eps_free_solve_tri*)
  have *length As* ≤ *length As'* **using** *2* **by** *simp*
  then have *Nts* (*expand_hd Aa As* (*solve_tri As As' R*)) ⊆ *Nts R* ∪ *set As'*
    using *2 Nts_expand_hd_sub Nts_solve_tri_sub*

```
    by (metis subset_trans)
  then have nts1:  A' ∉ Nts (expand_hd Aa As (solve_tri As As' R))
    using 2 Nts_expand_hd_sub Nts_solve_tri_sub by auto

  have Lang (solve_tri (Aa # As) (A' # As') R) A
        = Lang (solve_lrec Aa A' (expand_hd Aa As (solve_tri As As' R))) A
    by simp
  also have ... = Lang (expand_hd Aa As (solve_tri As As' R)) A
     using nts1 e_free1 2 solve_lrec_Lang[of expand_hd Aa As (solve_tri As As'
R) Aa A' A]
    by (simp)
  also have ... = Lang (solve_tri As As' R) A by (simp add: expand_hd_Lang)
  finally show ?case using 2 by (auto)
next
  case (3 v va c)
  then show ?case by simp
qed
```

5 Function *expand_hd*: Convert Triangular Form into GNF

5.1 *expand_hd*: Result is in *GNF_hd*

```
lemma dep_on_helper: dep_on R A = {} ⟹ (A, w) ∈ R ⟹ w = [] ∨ (∃ T wt.
w = Tm T # wt)
  using neq_Nil_conv[of w] by (simp add: dep_on_def) (metis sym.exhaust)

lemma GNF_hd_iff_dep_on:
  assumes Eps_free R
  shows GNF_hd R ⟷ (∀ A ∈ Nts R. dep_on R A = {}) (is ?L=?R)
proof
  assume ?L
  then show ?R by (auto simp add: GNF_hd_def dep_on_def)
next
  assume assm: ?R
  have 1: ∀ (B, w) ∈ R. ∃ T wt. w = Tm T # wt ∨ w = []
  proof
    fix x
    assume x ∈ R
    then have case x of (B, w) ⇒ dep_on R B = {} using assm by (auto simp
add: Nts_def)
    then show case x of (B, w) ⇒ ∃ T wt. w = Tm T # wt ∨ w = []
      using ‹x ∈ R› dep_on_helper by fastforce
  qed
 have 2: ∀ (B, w) ∈ R. w ≠ [] using assms assm by (auto simp add: Eps_free_def)
  have ∀ (B, w) ∈ R. ∃ T wt. w = Tm T # wt using 1 2 by auto
  then show GNF_hd R by (auto simp add: GNF_hd_def)
qed
```

lemma helper_expand_tri1: A ∉ set As ⟹ (A, w) ∈ expand_tri As R ⟹ (A, w) ∈ R
by (induction As R rule: expand_tri.induct) (auto simp add: Let_def)

If none of the expanded Nts depend on A then any rule depending on A in expand_tri As R must already have been in R.

lemma helper_expand_tri2:
⟦Eps_free R; A ∉ set As; ∀ C ∈ set As. A ∉ (dep_on R C); B ≠ A; (B, Nt A # w) ∈ expand_tri As R⟧
⟹ (B, Nt A # w) ∈ R
proof (induction As R arbitrary: B w rule: expand_tri.induct)
case (1 R)
then show ?case **by** simp
next
case (2 S Ss R)
have (B, Nt A # w) ∈ expand_tri Ss R
proof (cases B = S)
case B_is_S: True
let ?R′ = expand_tri Ss R
let ?X = {(Al,Bw) ∈ ?R′. Al=S ∧ (∃ w B. Bw = Nt B # w ∧ B ∈ set (Ss))}
let ?Y = {(S,v@w) |v w. ∃ B. (S, Nt B # w) ∈ ?X ∧ (B,v) ∈ ?R′}
have (B, Nt A # w) ∉ ?X **using** 2 **by** auto
then have 3: (B, Nt A # w) ∈ ?R′ ∨ (B, Nt A # w) ∈ ?Y **using** 2 **by** (auto simp add: Let_def)
then show ?thesis
proof (cases (B, Nt A # w) ∈ ?R′)
case True
then show ?thesis **by** simp
next
case False
then have (B, Nt A # w) ∈ ?Y **using** 3 **by** simp
then have ∃ v wa Ba. Nt A # w = v @ wa ∧ (S, Nt Ba # wa) ∈ expand_tri Ss R ∧ Ba ∈ set Ss
∧ (Ba, v) ∈ expand_tri Ss R
by (auto simp add: Let_def)
then obtain v wa Ba
where P: Nt A # w = v @ wa ∧ (S, Nt Ba # wa) ∈ expand_tri Ss R ∧ Ba ∈ set Ss
∧ (Ba, v) ∈ expand_tri Ss R
by blast
have Eps_free (expand_tri Ss R) **using** 2 **by** (auto simp add: Eps_free_expand_tri)
then have v ≠ [] **using** P **by** (auto simp add: Eps_free_def)
then have v_hd: hd v = Nt A **using** P **by** (metis hd_append list.sel(1))
then have ∃ va. v = Nt A # va
by (metis ‹v ≠ []› list.collapse)
then obtain va **where** P2: v = Nt A # va **by** blast
then have (Ba, v) ∈ R **using** 2 P
by (metis list.set_intros(2))
then have A ∈ dep_on R Ba **using** v_hd P2 **by** (auto simp add: dep_on_def)

then show *?thesis* **using** *2 P* **by** *auto*
qed
next
case *False*
then show *?thesis* **using** *2* **by** (*auto simp add*: *Let_def*)
qed

then show *?case* **using** *2* **by** *auto*
qed

In a triangular form no Nts depend on the last Nt in the list.

lemma *triangular_snoc_dep_on*: *triangular* (*As*@[*A*]) *R* $\Longrightarrow$ $\forall$ *C* $\in$ *set As*. *A* $\notin$ (*dep_on R C*)
by (*induction As*) *auto*

lemma *triangular_helper1*: *triangular As R* $\Longrightarrow$ *A* $\in$ *set As* $\Longrightarrow$ *A* $\notin$ *dep_on R A*
by (*induction As*) *auto*

lemma *dep_on_expand_tri*:
⟦*Eps_free R*; *triangular* (*rev As*) *R*; *distinct As*; *A* $\in$ *set As*⟧
$\Longrightarrow$ *dep_on* (*expand_tri As R*) *A* $\cap$ *set As* = {}
proof(*induction As R arbitrary*: *A rule*: *expand_tri.induct*)
case (*1 R*)
then show *?case* **by** *simp*
next
case (*2 S Ss R*)
then have *Eps_free_exp_Ss*: *Eps_free* (*expand_tri Ss R*)
by (*simp add*: *Eps_free_expand_tri*)
have *dep_on_fact*: $\forall$ *C* $\in$ *set Ss*. *S* $\notin$ (*dep_on R C*)
using *2* **by** (*auto simp add*: *triangular_snoc_dep_on*)
then show *?case*
proof (*cases A* = *S*)
case *True*
have *F1*: (*S*, *Nt S* # *w*) $\notin$ *expand_tri Ss R* **for** *w*
proof(*rule ccontr*)
assume ¬((*S*, *Nt S* # *w*) $\notin$ *expand_tri Ss R*)
then have (*S*, *Nt S* # *w*) $\in$ *R* **using** *2* **by** (*auto simp add*: *helper_expand_tri1*)
then have *N*: *S* $\in$ *dep_on R A* **using** *True* **by** (*auto simp add*: *dep_on_def*)
have *S* $\notin$ *dep_on R A* **using** *2 True* **by** (*auto simp add*: *triangular_helper1*)
then show *False* **using** *N* **by** *simp*
qed

have *F2*: (*S*, *Nt S* # *w*) $\notin$ *expand_tri* (*S*#*Ss*) *R* **for** *w*
proof
assume (*S*, *Nt S* # *w*) $\in$ *expand_tri* (*S*#*Ss*) *R*
then have $\exists$ *v wa B*. *Nt S* # *w* = *v* @ *wa* $\wedge$ *B* $\in$ *set Ss* $\wedge$ (*S*, *Nt B* # *wa*) $\in$ *expand_tri Ss R*
$\wedge$ (*B*, *v*) $\in$ *expand_tri Ss R*

```
      using 2 F1 by (auto simp add: Let_def)
    then obtain v wa B
      where v_wa_B_P: Nt S # w = v @ wa ∧ B ∈ set Ss ∧ (S, Nt B # wa)
∈ expand_tri Ss R
       ∧ (B, v) ∈ expand_tri Ss R
      by blast
    then have v ≠ [] ∧ (∃va. v = Nt S # va) using Eps_free_exp_Ss
      by (metis Eps_free_Nil append_eq_Cons_conv)
    then obtain va where vP: v ≠ [] ∧ v = Nt S # va by blast
    then have (B, v) ∈ R
       using v_wa_B_P 2 dep_on_fact helper_expand_tri2[of R S Ss B] True
by auto
    then have S ∈ dep_on R B using vP by (auto simp add: dep_on_def)
    then show False using dep_on_fact v_wa_B_P by auto
  qed

  have (S, Nt x # w) ∉ expand_tri (S#Ss) R if asm: x ∈ set Ss for x w
  proof
    assume assm: (S, Nt x # w) ∈ expand_tri (S # Ss) R
    then have ∃v wa B. Nt x # w = v @ wa ∧ (S, Nt B # wa) ∈ expand_tri
Ss R ∧ B ∈ set Ss
       ∧ (B, v) ∈ expand_tri Ss R
      using 2 asm by (auto simp add: Let_def)
    then obtain v wa B
       where v_wa_B_P:Nt x # w = v @ wa ∧ (S, Nt B # wa) ∈ expand_tri
Ss R ∧ B ∈ set Ss
       ∧ (B, v) ∈ expand_tri Ss R
      by blast
    then have dep_on_IH: dep_on (expand_tri Ss R) B ∩ set Ss = {}
      using 2 by (auto simp add: tri_Snoc_impl_tri)
    have v ≠ [] ∧ (∃va. v = Nt x # va) using Eps_free_exp_Ss v_wa_B_P
      by (metis Eps_free_Nil append_eq_Cons_conv)
    then obtain va where vP: v ≠ [] ∧ v = Nt x # va by blast
     then have x ∈ dep_on (expand_tri Ss R) B using v_wa_B_P by (auto
simp add: dep_on_def)
    then show False using dep_on_IH v_wa_B_P asm assm by auto
  qed

  then show ?thesis using 2 True F2 by (auto simp add: Let_def dep_on_def)
 next
  case False
  have (A, Nt S # w) ∉ expand_tri Ss R for w
  proof
    assume (A, Nt S # w) ∈ expand_tri Ss R
    then have (A, Nt S # w) ∈ R using 2 helper_expand_tri2 dep_on_fact
      by (metis False distinct.simps(2))
    then have F: S ∈ dep_on R A by (auto simp add: dep_on_def)
    have S ∉ dep_on R A using dep_on_fact False 2 by auto
    then show False using F by simp
```

qed
then show *?thesis* **using** *2 False* **by** (*auto simp add*: *tri_Snoc_impl_tri Let_def dep_on_def*)
qed
qed

Interlude: *Nts* of *expand_tri*:

lemma *Lhss_expand_tri*: *Lhss* (*expand_tri As R*) ⊆ *Lhss R*
by (*induction As R rule*: *expand_tri.induct*) (*auto simp add*: *Lhss_def Let_def*)

lemma *Rhs_Nts_expand_tri*: *Rhs_Nts* (*expand_tri As R*) ⊆ *Rhs_Nts R*
proof (*induction As R rule*: *expand_tri.induct*)
case (*1 R*)
then show *?case* **by** *simp*
next
case (*2 S Ss R*)
let *?X* = {(*Al*, *Bw*). (*Al*, *Bw*) ∈ *expand_tri Ss R* ∧ *Al* = *S* ∧ (∃ *w B*. *Bw* = *Nt B* # *w* ∧ *B* ∈ *set Ss*)}
let *?Y* = {(*S*,*v*@*w*)|*v w*. ∃ *B*. (*S*,*Nt B*#*w*) ∈ *expand_tri Ss R* ∧ *B* ∈ *set Ss* ∧ (*B*,*v*) ∈ *expand_tri Ss R*}
have *F1*: *Rhs_Nts ?X* ⊆ *Rhs_Nts R* **using** *2* **by** (*auto simp add*: *Rhs_Nts_def*)
have *Rhs_Nts ?Y* ⊆ *Rhs_Nts R*
proof
fix *x*
assume *x* ∈ *Rhs_Nts ?Y*
then have ∃ *y ys*. (*y*, *ys*) ∈ *?Y* ∧ *x* ∈ *nts_syms ys* **by** (*auto simp add*: *Rhs_Nts_def*)
then obtain *y ys* **where** *P1*: (*y*, *ys*) ∈ *?Y* ∧ *x* ∈ *nts_syms ys* **by** *blast*
then show *x* ∈ *Rhs_Nts R* **using** *P1 2 Rhs_Nts_def* **by** *fastforce*
qed
then show *?case* **using** *F1 2* **by** (*auto simp add*: *Rhs_Nts_def Let_def*)
qed

lemma *Nts_expand_tri*: *Nts* (*expand_tri As R*) ⊆ *Nts R*
by (*metis Lhss_expand_tri Nts_Lhss_Rhs_Nts Rhs_Nts_expand_tri Un_mono*)

If the entire *triangular* form is expanded, the result is in GNF:

theorem *GNF_hd_expand_tri*:
assumes *Eps_free R triangular* (*rev As*) *R distinct As Nts R* ⊆ *set As*
shows *GNF_hd* (*expand_tri As R*)
by (*metis Eps_free_expand_tri GNF_hd_iff_dep_on Int_absorb2 Nts_expand_tri assms dep_on_expand_tri dep_on_subs_Nts subset_trans subsetD*)

Any set of productions can be transformed into GNF via *expand_tri* (*solve_tri*).

theorem *GNF_of_R*:
assumes *assms*: *Eps_free R distinct* (*As* @ *As'*) *Nts R* ⊆ *set As length As* ≤ *length As'*
shows *GNF_hd* (*expand_tri* (*As'* @ *rev As*) (*solve_tri As As' R*))

proof –
  from *assms* **have** *tri*: *triangular* (*As* @ *rev As′*) (*solve_tri As As′ R*)
    by (*simp add*: *Int_commute triangular_As_As′_solve_tri*)
  have *Nts* (*solve_tri As As′ R*) ⊆ *set As* ∪ *set As′* **using** *assms Nts_solve_tri_sub* **by** *fastforce*
  then show *?thesis*
    using *GNF_hd_expand_tri*[*of* (*solve_tri As As′ R*) (*As′* @ *rev As*)] *assms tri*
    by (*auto simp add*: *Eps_free_solve_tri*)
qed

5.2 *expand_tri* Preserves Language

Similar to the proof of Language equivalence of *expand_hd*.

All productions in *expand_tri As R* are derivable by *R*.

lemma *expand_tri_prods_deirvable*: (*B*, *bs*) ∈ *expand_tri As R* ⟹ *R* ⊢ [*Nt B*] ⇒∗ *bs*
proof (*induction As R arbitrary*: *B bs rule*: *expand_tri.induct*)
  case (*1 R*)
  then show *?case*
    by (*simp add*: *bu_prod derives_if_bu*)
next
  case (*2 A As R*)
  then show *?case*
  proof (*cases B* ∈ *set* (*A*#*As*))
    case *True*
    then show *?thesis*
    proof (*cases B* = *A*)
      case *True*
      then have ∃ *C cw v*.(*bs* = *cw*@*v* ∧ (*B*, *Nt C*#*v*) ∈ (*expand_tri As R*) ∧ (*C*,*cw*) ∈ (*expand_tri As R*))
          ∨ (*B*, *bs*) ∈ (*expand_tri As R*)
        using *2* **by** (*auto simp add*: *Let_def*)
      then obtain *C cw v*
        where (*bs* = *cw* @ *v* ∧ (*B*, *Nt C* # *v*) ∈ (*expand_tri As R*) ∧ (*C*, *cw*) ∈ (*expand_tri As R*))
          ∨ (*B*, *bs*) ∈ (*expand_tri As R*)
        by *blast*
      then have (*bs* = *cw* @ *v* ∧ *R* ⊢ [*Nt B*] ⇒∗ [*Nt C*] @ *v* ∧ *R* ⊢ [*Nt C*] ⇒∗ *cw*) ∨ *R* ⊢ [*Nt B*] ⇒∗ *bs*
        using *2.IH* **by** *auto*
      then show *?thesis* **by** (*meson derives_append rtranclp_trans*)
    next
      case *False*
      then have (*B*, *bs*) ∈ (*expand_tri As R*) **using** *2* **by** (*auto simp add*: *Let_def*)
      then show *?thesis* **using** *2.IH* **by** (*simp add*: *bu_prod derives_if_bu*)
    qed
  next
    case *False*

```
    then have (B, bs) ∈ R using 2 by (auto simp only: helper_expand_tri1)
    then show ?thesis by (simp add: bu_prod derives_if_bu)
  qed
qed
```

Language Preservation:

```
lemma expand_tri_Lang: Lang (expand_tri As R) A = Lang R A
proof
  have (B, bs) ∈ (expand_tri As R) ⟹ R ⊢ [Nt B] ⇒* bs for B bs
    by (simp add: expand_tri_prods_deirvable)
  then have expand_tri As R ⊢ [Nt A] ⇒* map Tm x ⟹ R ⊢ [Nt A] ⇒* map Tm x for x
    using derives_simul_rules by blast
  then show Lang (expand_tri As R) A ⊆ Lang R A  by(auto simp add: Lang_def)
next
  show Lang R A ⊆ Lang (expand_tri As R) A
  proof (induction As R rule: expand_tri.induct)
    case (1 R)
    then show ?case by simp
  next
    case (2 D Ds R)
    let ?R' = expand_tri Ds R
    let ?X = {(Al,Bw) ∈ ?R'. Al=D ∧ (∃w B. Bw = Nt B # w ∧ B ∈ set (Ds))}
    let ?Y = {(D,v@w) |v w. ∃B. (D, Nt B # w) ∈ ?X ∧ (B,v) ∈ ?R'}
    have F1: expand_tri (D#Ds) R = ?R' − ?X ∪ ?Y by (simp add: Let_def)

    let ?S = {x. ∃A w H. x = (A, [], H, w) ∧ (A, Nt H # w) ∈ ?X}
    let ?S' = {x. ∃A a1 B a2. x = (A, a1 @ Nt B # a2) ∧ (A, a1, B, a2) ∈ ?S}
    let ?E = {x. ∃A v a1 a2 B. x = (A,a1@v@a2) ∧ (A, a1, B, a2) ∈ ?S ∧ (B,v) ∈ ?R'}

    have S'_eq_X: ?S' = ?X by fastforce
    have E_eq_Y: ?E = ?Y by fastforce

    have ∀x ∈ ?S. ∃A a1 B a2. x = (A, a1, B, a2) ∧ (A, a1 @ Nt B # a2) ∈ ?R' by fastforce
    have Lang R A ⊆ Lang (expand_tri Ds R) A using 2 by simp
    also have ... ⊆ Lang (?R' − ?S' ∪ ?E) A
      using exp_includes_Lang[of ?S] by auto
    also have ... = Lang (expand_tri (D#Ds) R) A using S'_eq_X E_eq_Y F1 by fastforce
    finally show ?case.
  qed
qed
```

6 Function *gnf_hd*: Conversion to *GNF_hd*

All epsilon-free grammars can be put into GNF while preserving their language.

Putting the productions into GNF via *expand_tri* (*solve_tri*) preserves the language.

lemma *GNF_of_R_Lang*:
assumes *Eps_free R length As* $\leq$ *length As′ distinct* (*As* @ *As′*) *Nts R* $\cap$ *set As′* = {} *A* $\notin$ *set As′*
shows *Lang* (*expand_tri* (*As′* @ *rev As*) (*solve_tri As As′ R*)) *A* = *Lang R A*
using *solve_tri_Lang*[*OF assms*] *expand_tri_Lang*[*of* (*As′* @ *rev As*)] **by** *blast*

Any epsilon-free Grammar can be brought into GNF.

theorem *GNF_hd_gnf_hd*: *eps_free ps* $\Longrightarrow$ *GNF_hd* (*gnf_hd ps*)
by(*simp add*: *gnf_hd_def Let_def GNF_of_R*[*simplified*]
distinct_nts_prods_list freshs_distinct finite_nts freshs_disj set_nts_prods_list length_freshs)

lemma *distinct_app_freshs*: ⟦ *As* = *nts_prods_list ps*; *As′* = *freshs* (*set As*) *As* ⟧ $\Longrightarrow$
distinct (*As* @ *As′*)
using *freshs_disj*[*of set As As*]
by (*auto simp*: *distinct_nts_prods_list freshs_distinct*)

gnf_hd preserves the language:

theorem *Lang_gnf_hd*: ⟦ *eps_free ps*; *A* $\in$ *nts ps* ⟧ $\Longrightarrow$ *Lang* (*gnf_hd ps*) *A* = *lang ps A*
unfolding *gnf_hd_def Let_def*
by (*metis GNF_of_R_Lang IntI distinct_app_freshs empty_iff finite_nts freshs_disj*

length_freshs order_refl set_nts_prods_list)

Two simple examples:

lemma *gnf_hd* [(*0*, [*Nt*(*0*::*nat*), *Tm* (*1*::*int*)])] = {(*1*, [*Tm 1*]), (*1*, [*Tm 1*, *Nt 1*])}
by *eval*

lemma *gnf_hd* [(*0*, [*Nt*(*0*::*nat*), *Tm* (*1*::*int*)]), (*0*, [*Tm 2*])] =
{ (*0*, [*Tm 2*, *Nt 1*]), (*0*, [*Tm 2*]), (*1*, [*Tm 1*, *Nt 1*]), (*1*, [*Tm 1*]) }
by *eval*

Example 4.10 [3]: *P0* is the input; *P1* is the result after Step 1; *P3* is the result after Step 2 and 3.

lemma
let
P0 =
[(*1*::*int*, [*Nt 2*, *Nt 3*]), (*2*,[*Nt 3*, *Nt 1*]), (*2*, [*Tm* (*1*::*int*)]), (*3*,[*Nt 1*, *Nt 2*]), (*3*,[*Tm 0*])];

P1 =
[(*1*, [*Nt 2*, *Nt 3*]), (*2*, [*Nt 3*, *Nt 1*]), (*2*, [*Tm 1*]),
(*3*, [*Tm 1*, *Nt 3*, *Nt 2*, *Nt 4*]), (*3*, [*Tm 0*, *Nt 4*]), (*3*, [*Tm 1*, *Nt 3*, *Nt 2*]), (*3*, [*Tm 0*]),
(*4*, [*Nt 1*, *Nt 3*, *Nt 2*]), (*4*, [*Nt 1*, *Nt 3*, *Nt 2*, *Nt 4*])];
P2 =
[(*1*, [*Tm 1*, *Nt 3*, *Nt 2*, *Nt 4*, *Nt 1*, *Nt 3*]), (*1*, [*Tm 1*, *Nt 3*, *Nt 2*, *Nt 1*, *Nt 3*]),
(*1*, [*Tm 0*, *Nt 4*, *Nt 1*, *Nt 3*]), (*1*, [*Tm 0*, *Nt 1*, *Nt 3*]), (*1*, [*Tm 1*, *Nt 3*]),
(*2*, [*Tm 1*, *Nt 3*, *Nt 2*, *Nt 4*, *Nt 1*]), (*2*, [*Tm 1*, *Nt 3*, *Nt 2*, *Nt 1*]),
(*2*, [*Tm 0*, *Nt 4*, *Nt 1*]), (*2*, [*Tm 0*, *Nt 1*]), (*2*, [*Tm 1*]),
(*3*, [*Tm 1*, *Nt 3*, *Nt 2*, *Nt 4*]), (*3*, [*Tm 1*, *Nt 3*, *Nt 2*]),
(*3*, [*Tm 0*, *Nt 4*]), (*3*, [*Tm 0*]),
(*4*, [*Tm 1*, *Nt 3*, *Nt 2*, *Nt 4*, *Nt 1*, *Nt 3*, *Nt 3*, *Nt 2*, *Nt 4*]), (*4*, [*Tm 1*, *Nt 3*, *Nt 2*, *Nt 4*, *Nt 1*, *Nt 3*, *Nt 3*, *Nt 2*]),
(*4*, [*Tm 0*, *Nt 4*, *Nt 1*, *Nt 3*, *Nt 3*, *Nt 2*, *Nt 4*]), (*4*, [*Tm 0*, *Nt 4*, *Nt 1*, *Nt 3*, *Nt 3*, *Nt 2*]),
(*4*, [*Tm 1*, *Nt 3*, *Nt 3*, *Nt 2*, *Nt 4*]), (*4*, [*Tm 1*, *Nt 3*, *Nt 3*, *Nt 2*]),
(*4*, [*Tm 1*, *Nt 3*, *Nt 2*, *Nt 1*, *Nt 3*, *Nt 3*, *Nt 2*, *Nt 4*]), (*4*, [*Tm 1*, *Nt 3*, *Nt 2*, *Nt 1*, *Nt 3*, *Nt 3*, *Nt 2*]),
(*4*, [*Tm 0*, *Nt 1*, *Nt 3*, *Nt 3*, *Nt 2*, *Nt 4*]), (*4*, [*Tm 0*, *Nt 1*, *Nt 3*, *Nt 3*, *Nt 2*])]
in
solve_tri [*3*,*2*,*1*] [*4*,*5*,*6*] (*set P0*) = *set P1* ∧ *expand_tri* [*4*,*1*,*2*,*3*] (*set P1*) = *set P2*
by *eval*

7 Complexity

Our method has exponential complexity, which we demonstrate below. Alternative polynomial methods are described in the literature [1].

We start with an informal proof that the blowup of the whole method can be as bad as 2^{n^2}, where n is the number of non terminals, and the starting grammar has $4n$ productions.

Consider this grammar, where a and b are terminals and we use nested alternatives in the obvious way:

A0 → *A1* (*a* | *b*) | *A2* (*a* | *b*) | ... | *An* (*a* | *b*) | *a* | *b*

A(*i*+*1*) → *Ai* (*a* | *b*)

Expanding all alternatives makes this a grammar of size $4n$.

When converting this grammar into triangular form, starting with *A0*, we find that *A0* remains the same after *expand_hd*, and *solve_lrec* introduces a new additional production for every *A0* production, which we will ignore to simplify things:

Then every *expand_hd* step yields for *Ai* these number of productions:

(1) *2*^(*i*+*1*) productions with rhs *Ak* (*a* | *b*)^(*i*+*1*) for every *k* ∈ [*i*+*1*, *n*],

(2) $2^{(i+1)}$ productions with rhs $(a \mid b)^{(i+1)}$,

(3) $2^{(i+1)}$ productions with rhs $Ai\ (a \mid b)^{(i+1)}$.

Note that $(a \mid b)^{(i+1)}$ represents all words of length $i+1$ over $\{a,b\}$. Solving the left recursion again introduces a new additional production for every production of form (1) and (2), which we will again ignore for simplicity. The productions of (3) get removed by *solve_lrec*. We will not consider the productions of the newly introduced nonterminals.

In the triangular form, every *Ai* has at least $2^{(i+1)}$ productions starting with terminals (2) and $2^{(i+1)}$ productions with rhs starting with *Ak* for every $k \in [i+1,\ n]$.

When expanding the triangular form starting from *An*, which has at least the $2^{(i+1)}$ productions from (2), we observe that the number of productions of *Ai* (denoted by $\#Ai$) is $\#Ai \geq 2^{(i+1)} * \#A(i+1)$ (Only considering the productions of the form $A(i+1)\ (a \mid b)^{(i+1)}$). This yields that $\#Ai \geq 2^{(i+1)} * 2^{((i+2) + ... + (n+1))} = 2^{((i+1) + (i+2) + ... (n+1))}$. Thus $\#A0 \geq 2^{(1 + 2 + ... + n + (n+1))} = 2^{((n+1)*(n+2)/2)}$.

Below we prove formally that *expand_tri* can cause exponential blowup.

Bad grammar: Constructs a grammar which leads to a exponential blowup when expanded by *expand_tri*:

fun *bad_grammar* :: *'n list* ⇒ (*'n*, *nat*)*Prods* **where**
 bad_grammar [] = {}
|*bad_grammar* [*A*] = {(*A*, [*Tm 0*]), (*A*, [*Tm 1*])}
|*bad_grammar* (*A*#*B*#*As*) = {(*A*, *Nt B* # [*Tm 0*]), (*A*, *Nt B* # [*Tm 1*])} ∪ (*bad_grammar* (*B*#*As*))

lemma *bad_gram_simp1*: *A* ∉ *set As* ⟹ (*A*, *Bs*) ∉ (*bad_grammar As*)
  by (*induction As rule*: *bad_grammar.induct*) *auto*

lemma *expand_tri_simp1*: *A* ∉ *set As* ⟹ (*A*, *Bs*) ∈ *R* ⟹ (*A*, *Bs*) ∈ *expand_tri As R*
  by (*induction As R rule*: *expand_tri.induct*) (*auto simp add*: *Let_def*)

lemma *expand_tri_iff1*: *A* ∉ *set As* ⟹ (*A*, *Bs*) ∈ *expand_tri As R* ⟷ (*A*, *Bs*) ∈ *R*
  using *expand_tri_simp1 helper_expand_tri1* **by** *auto*

lemma *expand_tri_insert_simp*:
  B ∉ *set As* ⟹ *expand_tri As* (*insert* (*B*, *Bs*) *R*) = *insert* (*B*, *Bs*) (*expand_tri As R*)
  by (*induction As R rule*: *expand_tri.induct*) (*auto simp add*: *Let_def*)

lemma *expand_tri_bad_grammar_simp1*:
  distinct (*A*#*As*) ⟹ *length As* ≥ *1*
  ⟹ *expand_tri As* (*bad_grammar* (*A*#*As*))
    = {(*A*, *Nt* (*hd As*) # [*Tm 0*]), (*A*, *Nt* (*hd As*) # [*Tm 1*])} ∪ (*expand_tri As*

```
(bad_grammar As))
proof (induction As)
  case Nil
  then show ?case by simp
next
  case Cons1: (Cons B Bs)
  then show ?case
  proof (cases Bs)
    case Nil
    then show ?thesis by auto
  next
    case Cons2: (Cons C Cs)
    then show ?thesis using Cons1 expand_tri_insert_simp
        by (smt (verit) Un_insert_left bad_grammar.elims distinct.simps(2) in-
sert_is_Un
          list.distinct(1) list.inject list.sel(1))
  qed
qed

lemma finite_bad_grammar: finite (bad_grammar As)
  by (induction As rule: bad_grammar.induct) auto

lemma finite_expand_tri: finite R ⟹ finite (expand_tri As R)
proof (induction As R rule: expand_tri.induct)
  case (1 R)
  then show ?case by simp
next
  case (2 S Ss R)
  let ?S = {(S,v@w)|v w. ∃ B. (S,Nt B#w) ∈ expand_tri Ss R ∧ B ∈ set Ss ∧
(B,v) ∈ expand_tri Ss R}
  let ?f = λ((A,w),(B,v)). (A, v @ (tl w))
  have ?S ⊆ ?f ‘ ((expand_tri Ss R) × (expand_tri Ss R))
  proof
    fix x
    assume x ∈ ?S
    then have ∃ S v B w. (S,Nt B # w) ∈ expand_tri Ss R ∧ (B,v) ∈ expand_tri
Ss R ∧ x = (S, v @ w)
      by blast
    then obtain S v B w
      where P: (S, Nt B # w) ∈ expand_tri Ss R ∧ (B, v) ∈ expand_tri Ss R ∧
x = (S, v @ w)
      by blast
    then have 1: ((S, Nt B # w), (B ,v)) ∈ ((expand_tri Ss R) × (expand_tri Ss
R)) by auto
    have ?f ((S, Nt B # w), (B ,v)) = (S, v @ w) by auto
    then have (S, v @ w) ∈ ?f ‘ ((expand_tri Ss R) × (expand_tri Ss R)) using
1 by force
   then show x ∈ ?f ‘ ((expand_tri Ss R) × (expand_tri Ss R)) using P by simp
  qed
```

then have *finite ?S*
by (*meson 2.IH 2.prems finite_SigmaI finite_surj*)
then show *?case* **using** *2* **by** (*auto simp add*: *Let_def*)
qed

The last Nt expanded by *expand_tri* has an exponential number of productions.

lemma *bad_gram_last_expanded_card*:
⟦*distinct As*; *length As* = *n*; *n* ≥ *1*⟧
⟹ *card* ({*v*. (*hd As*, *v*) ∈ *expand_tri As* (*bad_grammar As*)}) = *2* ^ *n*
proof(*induction As arbitrary*: *n rule*: *bad_grammar.induct*)
case *1*
then show *?case* **by** *simp*
next
case (*2 A*)
have *4*: {*v*. *v* = [*Tm 0*] ∨ *v* = [*Tm* (*Suc 0*)]} = {[*Tm 0*], [*Tm 1*]} **by** *auto*
then show *?case* **using** *2* **by** (*auto simp add*: *4*)
next
case (*3 A C As*)
let *?R′* = *expand_tri* (*C#As*) (*bad_grammar* (*A#C#As*))
let *?X* = {(*Al*,*Bw*) ∈ *?R′*. *Al*=*A* ∧ (∃ *w B*. *Bw* = *Nt B* # *w* ∧ *B* ∈ *set* (*C#As*))}
let *?Y* = {(*A*,*v*@*w*) |*v w*. ∃ *B*. (*A*, *Nt B* # *w*) ∈ *?X* ∧ (*B*,*v*) ∈ *?R′*}

let *?S* = {*v*. (*hd* (*A#C#As*), *v*) ∈ *expand_tri* (*A#C#As*) (*bad_grammar* (*A#C#As*))}

have *4*: (*A*,*Bw*) ∈ *?R′* ⟷ (*A*, *Bw*) ∈ (*bad_grammar* (*A#C#As*)) **for** *Bw*
using *expand_tri_iff1*[*of A C#As Bw*] *3* **by** *auto*
then have *?X* = {(*Al*,*Bw*) ∈ (*bad_grammar* (*A#C#As*)). *Al*=*A* ∧ (∃ *w B*. *Bw* = *Nt B#w* ∧ *B* ∈ *set* (*C#As*))}
using *expand_tri_iff1* **by** *auto*
also have ... = {(*A*, *Nt C* # [*Tm 0*]), (*A*, *Nt C* # [*Tm 1*])}
using *3* **by** (*auto simp add*: *bad_gram_simp1*)
finally have *5*: *?X* = {(*A*, [*Nt C*, *Tm 0*]), (*A*, [*Nt C*, *Tm 1*])}**.**
then have *cons5*: *?X* = {(*A*, *Nt C* # [*Tm 0*]), (*A*, *Nt C* # [*Tm 1*])} **by** *simp*

have *6*: *?R′* = {(*A*, [*Nt C*, *Tm 0*]), (*A*, [*Nt C*, *Tm 1*])} ∪ *expand_tri* (*C#As*) (*bad_grammar* (*C#As*))
using *3 expand_tri_bad_grammar_simp1*[*of A C#As*] **by** *auto*
have *8*: (*A*, *as*) ∉ *expand_tri* (*C#As*) (*bad_grammar* (*C#As*)) **for** *as*
using *3.prems bad_gram_simp1 expand_tri_iff1*
by (*metis distinct.simps*(*2*))
then have *7*: {(*A*,[*Nt C*, *Tm 0*]), (*A*,[*Nt C*, *Tm 1*])} ∩ *expand_tri* (*C#As*) (*bad_grammar* (*C#As*)) = {}
by *auto*

have *?R′* − *?X* = *expand_tri* (*C#As*) (*bad_grammar* (*C#As*)) **using** *7 6 5* **by** *auto*
then have *S_from_Y*: *?S* = {*v*. (*A*, *v*) ∈ *?Y*} **using** *6 8* **by** *auto*

have *Y_decomp*: *?Y* = {(*A*, *v* @ [*Tm 0*]) | *v*. (*C*,*v*) ∈ *?R'*} ∪ {(*A*, *v* @ [*Tm 1*]) | *v*. (*C*,*v*) ∈ *?R'*}
proof
show *?Y* ⊆ {(*A*, *v* @ [*Tm 0*]) | *v*. (*C*,*v*) ∈ *?R'*} ∪ {(*A*, *v* @ [*Tm 1*]) | *v*. (*C*,*v*) ∈ *?R'*}
proof
fix *x*
assume *assm*: *x* ∈ *?Y*
then have ∃ *v w*. *x* = (*A*, *v* @ *w*) ∧ (∃ *B*. (*A*, *Nt B* # *w*) ∈ *?X* ∧ (*B*,*v*) ∈ *?R'*) **by** *blast*
then obtain *v w* **where** *P*: *x* = (*A*, *v* @ *w*) ∧ (∃ *B*. (*A*, *Nt B* # *w*) ∈ *?X* ∧ (*B*,*v*) ∈ *?R'*) **by** *blast*
then have *cfact*:(*A*, *Nt C* # *w*) ∈ *?X* ∧ (*C*,*v*) ∈ *?R'* **using** *cons5*
by (*metis* (*no_types*, *lifting*) *Pair_inject insert_iff list.inject singletonD sym.inject*(*1*))
then have *w* = [*Tm 0*] ∨ *w* = [*Tm 1*] **using** *cons5*
by (*metis* (*no_types*, *lifting*) *empty_iff insertE list.inject prod.inject*)
then show *x* ∈ {(*A*, *v* @ [*Tm 0*]) | *v*. (*C*,*v*) ∈ *?R'*} ∪ {(*A*, *v* @ [*Tm 1*]) | *v*. (*C*,*v*) ∈ *?R'*}
using *P cfact* **by** *auto*
qed
next
show {(*A*, *v* @ [*Tm 0*]) | *v*. (*C*,*v*) ∈ *?R'*} ∪ {(*A*, *v* @ [*Tm 1*]) | *v*. (*C*,*v*) ∈ *?R'*} ⊆ *?Y*
using *cons5* **by** *auto*
qed

from *Y_decomp* **have** *S_decomp*: *?S* = {*v*@[*Tm 0*] | *v*. (*C*, *v*) ∈ *?R'*} ∪ {*v*@[*Tm 1*] | *v*. (*C*, *v*) ∈ *?R'*}
using *S_from_Y* **by** *auto*

have *cardCvR*: *card* {*v*. (*C*, *v*) ∈ *?R'*} = *2*^(*n*−*1*) **using** *3 6* **by** *auto*
have *bij_betw* (λ*x*. *x*@[*Tm 0*]) {*v*. (*C*, *v*) ∈ *?R'*} {*v*@[*Tm 0*] | *v*. (*C*, *v*) ∈ *?R'*}
by (*auto simp add*: *bij_betw_def inj_on_def*)
then have *cardS1*: *card* {*v*@[*Tm 0*] | *v*. (*C*, *v*) ∈ *?R'*} = *2*^(*n*−*1*)
using *cardCvR* **by** (*auto simp add*: *bij_betw_same_card*)
have *bij_betw* (λ*x*. *x*@[*Tm 1*]) {*v*. (*C*, *v*) ∈ *?R'*} {*v*@[*Tm 1*] | *v*. (*C*, *v*) ∈ *?R'*}
by (*auto simp add*: *bij_betw_def inj_on_def*)
then have *cardS2*: *card* {*v*@[*Tm 1*] | *v*. (*C*, *v*) ∈ *?R'*} = *2*^(*n*−*1*)
using *cardCvR* **by** (*auto simp add*: *bij_betw_same_card*)

have *fin_R'*: *finite ?R'* **using** *finite_bad_grammar finite_expand_tri* **by** *blast*
let *?f1* = λ(*C*,*v*). *v*@[*Tm 0*]
have {*v*@[*Tm 0*] | *v*. (*C*, *v*) ∈ *?R'*} ⊆ *?f1* ‘ *?R'* **by** *auto*
then have *fin1*: *finite* {*v*@[*Tm 0*] | *v*. (*C*, *v*) ∈ *?R'*}
using *fin_R'* **by** (*meson finite_SigmaI finite_surj*)
let *?f2* = λ(*C*,*v*). *v*@[*Tm 1*]
have {*v*@[*Tm 1*] | *v*. (*C*, *v*) ∈ *?R'*} ⊆ *?f2* ‘ *?R'* **by** *auto*

then have fin2: finite {v@[Tm 1] | v. (C, v) ∈ ?R′}
  using fin_R′ **by** (meson finite_SigmaI finite_surj)

have fin_sets: finite {v@[Tm 0] | v. (C, v) ∈ ?R′} ∧ finite {v@[Tm 1] | v. (C, v) ∈ ?R′}
  using fin1 fin2 **by** simp

have {v@[Tm 0] | v. (C, v) ∈ ?R′} ∩ {v@[Tm 1] | v. (C, v) ∈ ?R′} = {} **by** auto
then have card ?S = 2^(n−1) + 2^(n−1)
  using S_decomp cardS1 cardS2 fin_sets
  by (auto simp add: card_Un_disjoint)

then show ?case **using** 3 **by** auto
qed

The productions resulting from *expand_tri* (*bad_grammar*) have at least exponential size.

theorem expand_tri_blowup: **assumes** n ≥ 1
  shows card (expand_tri [0..<n] (bad_grammar [0..<n])) ≥ 2^n
proof −
  from assms **have** length [0..<n] ≥ 1 ∧ distinct [0..<n] ∧ length [0..<n] = n **by** auto
  then have 1: card ({v. (hd [0..<n], v) ∈ expand_tri [0..<n] (bad_grammar [0..<n])}) = 2 ^ n
    using bad_gram_last_expanded_card assms **by** blast

  let ?S = {v. (hd [0..<n], v) ∈ expand_tri [0..<n] (bad_grammar [0..<n])}
  have 2: card ?S = card ({hd [0..<n]} × ?S)
    by (simp add: card_cartesian_product_singleton)
  have 3: ({hd [0..<n]} × ?S) ⊆ (expand_tri [0..<n] (bad_grammar [0..<n])) **by** fastforce

  have finite (expand_tri [0..<n] (bad_grammar [0..<n]))
    using finite_bad_grammar finite_expand_tri **by** blast
  then show ?thesis **using** 1 2 3
    by (metis card_mono)
qed

end

References

[1] N. Blum and R. Koch. Greibach normal form transformation revisited. *Inf. Comput.*, 150(1):112–118, 1999.

[2] S. A. Greibach. A new normal-form theorem for context-free phrase structure grammars. *J. ACM*, 12(1):42–52, 1965.

[3] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.