Context-Free Grammars and Languages

Tobias Nipkow, Markus Gschoßmann, Felix Krayer, Fabian Lehr,
Bruno Philipp, August Martin Stimpfle, Kaan Taskin, Akihisa Yamada

October 13, 2025

Abstract

This is a basic library of definitions and results about context-free grammars and languages. It includes context-free grammars and languages, parse trees, Chomsky normal form, pumping lemmas and the relationship of right-linear grammars to finite automata.

Contents

1 Context-Free Grammars

theory *Context_Free_Grammar*
imports *Fresh_Identifiers.Fresh_Nat*
begin

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

lemma *bex_pair_conv*: $(\exists (x,y) \in R.\ P\ x\ y) \longleftrightarrow (\exists x\ y.\ (x,y) \in R \wedge P\ x\ y)$
by *auto*

lemma *in_image_map_prod*: *fgp* $\in$ *map_prod f g* ‘ *R* $\longleftrightarrow$ $(\exists (x,y) \in R.\ fgp = (f\ x, g\ y))$
by *auto*

1.1 Symbols and Context-Free Grammars

Most of the theory is based on arbitrary sets of productions. Finiteness of the set of productions is only required where necessary. Finiteness of the type of terminal symbols is only required where necessary. Whenever fresh nonterminals need to be invented, the type of nonterminals is assumed to be infinite.

datatype $('n,'t)$ *sym* = *Nt* $'n$ | *Tm* $'t$

type_synonym $('n,'t)$ *syms* = $('n,'t)$ *sym list*

type_synonym $('n,'t)$ *prod* = $'n \times ('n,'t)$ *syms*

type_synonym $('n,'t)$ *prods* = $('n,'t)$ *prod list*
type_synonym $('n,'t)$ *Prods* = $('n,'t)$ *prod set*

datatype $('n,'t)$ *cfg* = *cfg* (*prods* : $('n,'t)$ *prods*) (*start* : $'n$)
datatype $('n,'t)$ *Cfg* = *Cfg* (*Prods* : $('n,'t)$ *Prods*) (*Start* : $'n$)

definition *isTm* :: $('n,\ 't)$ *sym* $\Rightarrow$ *bool* **where**
isTm S = $(\exists a.\ S = Tm\ a)$

definition *isNt* :: $('n,\ 't)$ *sym* $\Rightarrow$ *bool* **where**
isNt S = $(\exists A.\ S = Nt\ A)$

fun *destTm* :: $('n,\ 't)$ *sym* $\Rightarrow$ $'t$ **where**
‹*destTm* (*Tm a*) = *a*›

lemma *isTm_simps*[*simp*]:
‹*isTm* (*Nt A*) = *False*›
‹*isTm* (*Tm a*)›
by (*simp_all add*: *isTm_def*)

lemma *filter_isTm_map_Tm*[*simp*]: ‹*filter isTm* (*map Tm xs*) = *map Tm xs*›

by(*induction xs*) *auto*

lemma *destTm_o_Tm*[*simp*]: ‹*destTm* ∘ *Tm* = *id*›
by *auto*

definition *nts_syms* :: ('*n*,'*t*)*syms* ⇒ '*n set* **where**
nts_syms w = {*A*. *Nt A* ∈ *set w*}

lemma *nts_syms_code*[*code*]: *nts_syms w* = (⋃ *s* ∈ *set w*. *case s of Nt A* ⇒ {*A*}
| _ ⇒ {})
unfolding *nts_syms_def* **by** (*auto split*: *sym.splits*)

definition *tms_syms* :: ('*n*,'*t*)*syms* ⇒ '*t set* **where**
tms_syms w = {*a*. *Tm a* ∈ *set w*}

lemma *tms_syms_code*[*code*]: *tms_syms w* = (⋃ *s* ∈ *set w*. *case s of Tm a* ⇒ {*a*}
| _ ⇒ {})
unfolding *tms_syms_def* **by** (*auto split*: *sym.splits*)

definition *Nts* :: ('*n*,'*t*)*Prods* ⇒ '*n set* **where**
Nts P = (⋃(*A*,*w*)∈*P*. {*A*} ∪ *nts_syms w*)

definition *Tms* :: ('*n*,'*t*)*Prods* ⇒ '*t set* **where**
Tms P = (⋃(*A*,*w*)∈*P*. *tms_syms w*)

abbreviation *nts* :: ('*n*,'*t*) *prods* ⇒ '*n set* **where**
nts P ≡ *Nts* (*set P*)

definition *Syms* :: ('*n*,'*t*)*Prods* ⇒ ('*n*,'*t*) *sym set* **where**
Syms P = (⋃(*A*,*w*)∈*P*. {*Nt A*} ∪ *set w*)

abbreviation *tms* :: ('*n*,'*t*) *prods* ⇒ '*t set* **where**
tms P ≡ *Tms* (*set P*)

abbreviation *syms* :: ('*n*,'*t*) *prods* ⇒ ('*n*,'*t*) *sym set* **where**
syms P ≡ *Syms* (*set P*)

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* []

definition *Lhss* :: ('*n*, '*t*) *Prods* ⇒ '*n set* **where**
Lhss P = (⋃(*A*,*w*) ∈ *P*. {*A*})

abbreviation *lhss* :: ('*n*, '*t*) *prods* ⇒ '*n set* **where**
lhss ps ≡ *Lhss*(*set ps*)

definition *Rhs_Nts* :: (*'n*, *'t*) *Prods* ⇒ *'n set* **where**
Rhs_Nts P = (⋃(_,*w*)∈*P*. *nts_syms w*)

definition *Rhss* :: (*'n* × *'a*) *set* ⇒ *'n* ⇒ *'a set* **where**
Rhss P A = {*w*. (*A*,*w*) ∈ *P*}

lemma *inj_Nt*: *inj Nt*
by (*simp add*: *inj_def*)

lemma *map_Tm_inject_iff*[*simp*]: *map Tm xs* = *map Tm ys* ⟷ *xs* = *ys*
by (*metis sym.inject*(*2*) *list.inj_map_strong*)

lemma *map_Nt_eq_map_Nt_iff*[*simp*]: *map Nt u* = *map Nt v* ⟷ *u* = *v*
by(*rule inj_map_eq_map*[*OF inj_Nt*])

lemma *map_Nt_eq_map_Tm_iff*[*simp*]: *map Nt u* = *map Tm v* ⟷ *u* = [] ∧ *v* = []
by (*cases u*) *auto*

lemmas *map_Tm_eq_map_Nt_iff*[*simp*] = *eq_iff_swap*[*OF map_Nt_eq_map_Tm_iff*]

lemma *nts_syms_Nil*[*simp*]: *nts_syms* [] = {}
unfolding *nts_syms_def* **by** *auto*

lemma *nts_syms_Cons*[*simp*]: *nts_syms* (*a*#*v*) = (*case a of Nt A* ⇒ {*A*} | _ ⇒ {}) ∪ *nts_syms v*
by (*auto simp*: *nts_syms_def split*: *sym.split*)

lemma *nts_syms_append*[*simp*]: *nts_syms* (*u* @ *v*) = *nts_syms u* ∪ *nts_syms v*
by (*auto simp*: *nts_syms_def*)

lemma *nts_syms_map_Nt*[*simp*]: *nts_syms* (*map Nt w*) = *set w*
unfolding *nts_syms_def* **by** *auto*

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

lemma *nts_syms_map_Tm*[*simp*]: *nts_syms* (*map Tm w*) = {}
unfolding *nts_syms_def* **by** *auto*

lemma *nts_syms_empty_iff*: *nts_syms w* = {} ⟷ (∃ *u*. *w* = *map Tm u*)
by(*induction w*) (*auto simp*: *ex_map_conv split*: *sym.split*)

If a sentential form contains a *Nt*, it must have a last and a first *Nt*:

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 *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*)

lemma *in_Nts_iff_in_Syms*: *B* ∈ *Nts P* ⟷ *Nt B* ∈ *Syms P*
unfolding *Nts_def Syms_def nts_syms_def* **by** (*auto*)

lemma *Nts_mono*: *G* ⊆ *H* ⟹ *Nts G* ⊆ *Nts H*
by (*auto simp add*: *Nts_def*)

lemma *Nts_Un*: *Nts* (*P1* ∪ *P2*) = *Nts P1* ∪ *Nts P2*
by (*simp add*: *Nts_def*)

lemma *Nts_Lhss_Rhs_Nts*: *Nts P* = *Lhss P* ∪ *Rhs_Nts P*
unfolding *Nts_def Lhss_def Rhs_Nts_def* **by** *auto*

lemma *Nts_nts_syms*: *w* ∈ *Rhss P A* ⟹ *nts_syms w* ⊆ *Nts P*
unfolding *Rhss_def Nts_def* **by** *blast*

lemma *Syms_simps*[*simp*]:
Syms {} = {}
Syms(*insert* (*A*,*w*) *P*) = {*Nt A*} ∪ *set w* ∪ *Syms P*
Syms(*P* ∪ *P'*) = *Syms P* ∪ *Syms P'*
by(*auto simp*: *Syms_def*)

lemma *Lhss_simps*[*simp*]:
Lhss {} = {}
Lhss(*insert* (*A*,*w*) *P*) = {*A*} ∪ *Lhss P*
Lhss(*P* ∪ *P'*) = *Lhss P* ∪ *Lhss P'*
by(*auto simp*: *Lhss_def*)

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*)

1.1.1 Finiteness Lemmas

lemma *finite_nts_syms*: *finite* (*nts_syms w*)
proof −
have *Nt* ' {*A*. *Nt A* ∈ *set w*} ⊆ *set w* **by** *auto*
from *finite_inverse_image*[*OF* _ *inj_Nt*]
show *?thesis* **unfolding** *nts_syms_def* **using** *finite_inverse_image*[*OF* _ *inj_Nt*]
by *auto*
qed

lemma *finite_nts*: *finite*(*nts ps*)
unfolding *Nts_def* **by** (*simp add*: *finite_nts_syms split_def*)

lemma *fresh0_nts*: *fresh0*(*nts ps*) ∉ *nts ps*
by(*fact fresh0_notIn*[*OF finite_nts*])

lemma *finite_nts_prods_start*: *finite*(*nts*(*prods g*) ∪ {*start g*})
unfolding *Nts_def* **by** (*simp add*: *finite_nts_syms split_def*)

lemma *fresh_nts_prods_start*: *fresh0*(*nts*(*prods g*) ∪ {*start g*}) ∉ *nts*(*prods g*) ∪
{*start g*}
by(*fact fresh0_notIn*[*OF finite_nts_prods_start*])

lemma *finite_Nts*: *finite P* ⟹ *finite* (*Nts P*)
unfolding *Nts_def* **by** (*simp add*: *case_prod_beta finite_nts_syms*)

lemma *finite_Rhss*: *finite P* ⟹ *finite* (*Rhss P A*)
unfolding *Rhss_def* **by** (*metis Image_singleton finite_Image*)

1.2 Derivations

1.2.1 The standard derivations ⇒, ⇒∗, ⇒(*n*)

inductive *derive* :: ('*n*,'*t*) *Prods* ⇒ ('*n*,'*t*) *syms* ⇒ ('*n*,'*t*)*syms* ⇒ *bool*
((2_ ⊢/ (_ ⇒/ _)) [50, 0, 50] 50) **where**
(*A*,*α*) ∈ *P* ⟹ *P* ⊢ *u* @ [*Nt A*] @ *v* ⇒ *u* @ *α* @ *v*

abbreviation *deriven* ((2_ ⊢/ (_ /⇒'(_')/ _)) [50, 0, 0, 50] 50) **where**
P ⊢ *u* ⇒(*n*) *v* ≡ (*derive P* ^^ *n*) *u v*

abbreviation *derives* ((2_ ⊢/ (_/ ⇒∗/ _)) [50, 0, 50] 50) **where**
P ⊢ *u* ⇒∗ *v* ≡ ((*derive P*) ^∗∗) *u v*

definition *Ders* :: ('*n*,'*t*)*Prods* ⇒ '*n* ⇒ ('*n*,'*t*)*syms set* **where**

Ders P A = {*w*. *P* ⊢ [*Nt A*] ⇒∗ *w*}

abbreviation *ders* :: ('*n*,'*t*)*prods* ⇒ '*n* ⇒ ('*n*,'*t*)*syms set* **where**
ders ps ≡ *Ders* (*set ps*)

lemma *DersI*:
assumes *P* ⊢ [*Nt A*] ⇒∗ *w* **shows** *w* ∈ *Ders P A*
using *assms* **by** (*auto simp*: *Ders_def*)

lemma *DersD*:
assumes *w* ∈ *Ders P A* **shows** *P* ⊢ [*Nt A*] ⇒∗ *w*
using *assms* **by** (*auto simp*: *Ders_def*)

lemmas *DersE* = *DersD*[*elim_format*]

definition *Lang* :: ('*n*,'*t*)*Prods* ⇒ '*n* ⇒ '*t list set* **where**
Lang P A = {*w*. *P* ⊢ [*Nt A*] ⇒∗ *map Tm w*}

abbreviation *lang* :: ('*n*,'*t*)*prods* ⇒ '*n* ⇒ '*t list set* **where**
lang ps A ≡ *Lang* (*set ps*) *A*

abbreviation *LangS* :: ('*n*,'*t*) *Cfg* ⇒ '*t list set* **where**
LangS G ≡ *Lang* (*Prods G*) (*Start G*)

abbreviation *langS* :: ('*n*,'*t*) *cfg* ⇒ '*t list set* **where**
langS g ≡ *lang* (*prods g*) (*start g*)

lemma *Lang_Ders*: *map Tm* ' (*Lang P A*) ⊆ *Ders P A*
unfolding *Lang_def Ders_def* **by** *auto*

lemma *Lang_subset_if_Ders_subset*: *Ders R A* ⊆ *Ders R*′ *A* ⟹ *Lang R A* ⊆ *Lang R*′ *A*
by (*auto simp add*: *Lang_def Ders_def*)

lemma *Lang_eqI_derives*:
assumes ⋀*v*. *R* ⊢ [*Nt A*] ⇒∗ *map Tm v* ⟷ *S* ⊢ [*Nt A*] ⇒∗ *map Tm v*
shows *Lang R A* = *Lang S A*
by (*auto simp*: *Lang_def assms*)

lemma *derive_iff*: *R* ⊢ *u* ⇒ *v* ⟷ (∃ (*A*,*w*) ∈ *R*. ∃ *u1 u2*. *u* = *u1* @ *Nt A* # *u2* ∧ *v* = *u1* @ *w* @ *u2*)
apply (*rule iffI*)
apply (*induction rule*: *derive.induct*)
apply (*fastforce*)
using *derive.intros* **by** *fastforce*

lemma *not_derive_from_Tms*: ¬ *P* ⊢ *map Tm as* ⇒ *w*
by(*auto simp add*: *derive_iff map_eq_append_conv*)

lemma *deriven_from_TmsD*: $P \vdash map\ Tm\ as \Rightarrow(n)\ w \Longrightarrow w = map\ Tm\ as$
by (*metis not_derive_from_Tms relpowp_E2*)

lemma *derives_from_Tms_iff*: $P \vdash map\ Tm\ as \Rightarrow* w \longleftrightarrow w = map\ Tm\ as$
by (*meson deriven_from_TmsD rtranclp.rtrancl_refl rtranclp_power*)

lemma *Un_derive*: $R \cup S \vdash y \Rightarrow z \longleftrightarrow R \vdash y \Rightarrow z \vee S \vdash y \Rightarrow z$
by (*fastforce simp*: *derive_iff*)

lemma *derives_rule*:
assumes *2*: $(A,w) \in R$ **and** *1*: $R \vdash x \Rightarrow* y\ @\ Nt\ A\ \#\ z$ **and** *3*: $R \vdash y@w@z \Rightarrow* v$
shows $R \vdash x \Rightarrow* v$
proof−
note *1*
also have $R \vdash y\ @\ Nt\ A\ \#\ z \Rightarrow y\ @\ w\ @\ z$
using *derive.intros*[*OF 2*] **by** *simp*
also note *3*
finally show *?thesis*.
qed

lemma *derives_Cons_rule*:
assumes *1*: $(A,w) \in R$ **and** *2*: $R \vdash w\ @\ u \Rightarrow* v$ **shows** $R \vdash Nt\ A\ \#\ u \Rightarrow* v$
using *derives_rule*[*OF 1*, *of Nt A # u [] u v*] *2* **by** *auto*

lemma *deriven_mono*: $P \subseteq P' \Longrightarrow P \vdash u \Rightarrow(n)\ v \Longrightarrow P' \vdash u \Rightarrow(n)\ v$
by (*metis Un_derive relpowp_mono subset_Un_eq*)

lemma *derives_mono*: $P \subseteq P' \Longrightarrow P \vdash u \Rightarrow* v \Longrightarrow P' \vdash u \Rightarrow* v$
by (*meson deriven_mono rtranclp_power*)

lemma *Lang_mono*: $P \subseteq P' \Longrightarrow Lang\ P\ A \subseteq Lang\ P'\ A$
by (*auto simp*: *Lang_def derives_mono*)

1.2.2 Customized Induction Principles

lemma *deriven_induct*[*consumes 1*, *case_names 0 Suc*]:
assumes $P \vdash xs \Rightarrow(n)\ ys$
and $Q\ 0\ xs$
and $\bigwedge n\ u\ A\ v\ w.\ [\![\ P \vdash xs \Rightarrow(n)\ u\ @\ [Nt\ A]\ @\ v;\ Q\ n\ (u\ @\ [Nt\ A]\ @\ v);\ (A,w) \in P\ ]\!] \Longrightarrow Q\ (Suc\ n)\ (u\ @\ w\ @\ v)$
shows $Q\ n\ ys$
using *assms*(*1*) **proof** (*induction n arbitrary*: *ys*)
case *0*
thus *?case* **using** *assms*(*2*) **by** *auto*
next
case (*Suc n*)
from *relpowp_Suc_E*[*OF Suc.prems*]
obtain *xs'* **where** *n*: $P \vdash xs \Rightarrow(n)\ xs'$ **and** *1*: $P \vdash xs' \Rightarrow ys$ **by** *auto*

from *derive.cases*[*OF 1*] **obtain** *u A v w* **where** $xs' = u @ [Nt\ A] @ v$ $(A,w) \in P$ $ys = u @ w @ v$
by *metis*
with *Suc.IH*[*OF n*] *assms*(*3*) *n*
show *?case* **by** *blast*
qed

lemma *derives_induct*[*consumes 1*, *case_names base step*]:
assumes $P \vdash xs \Rightarrow* ys$
and *Q xs*
and $\bigwedge u\ A\ v\ w.\ [\![P \vdash xs \Rightarrow* u @ [Nt\ A] @ v;\ Q\ (u @ [Nt\ A] @ v);\ (A,w) \in P]\!] \Longrightarrow Q\ (u @ w @ v)$
shows *Q ys*
using *assms*
proof (*induction rule*: *rtranclp_induct*)
case *base*
from *this*(*1*) **show** *?case* **.**
next
case *step*
from *derive.cases*[*OF step*(*2*)] *step*(*1*,*3*−) **show** *?case* **by** *metis*
qed

lemma *converse_derives_induct*[*consumes 1*, *case_names base step*]:
assumes $P \vdash xs \Rightarrow* ys$
and *Base*: *Q ys*
and *Step*: $\bigwedge u\ A\ v\ w.\ [\![P \vdash u @ [Nt\ A] @ v \Rightarrow* ys;\ Q\ (u @ w @ v);\ (A,w) \in P]\!] \Longrightarrow Q\ (u @ [Nt\ A] @ v)$
shows *Q xs*
using *assms*(*1*)
apply (*induction rule*: *converse_rtranclp_induct*)
by (*auto elim*!: *derive.cases intro*!: *Base Step intro*: *derives_rule*)

1.2.3 (De)composing derivations

lemma *derive_append*:
$\mathcal{G} \vdash u \Rightarrow v \Longrightarrow \mathcal{G} \vdash u@w \Rightarrow v@w$
apply(*erule derive.cases*)
using *derive.intros* **by** *fastforce*

lemma *derive_prepend*:
$\mathcal{G} \vdash u \Rightarrow v \Longrightarrow \mathcal{G} \vdash w@u \Rightarrow w@v$
apply(*erule derive.cases*)
by (*metis append.assoc derive.intros*)

lemma *deriven_append*:
$P \vdash u \Rightarrow(n)\ v \Longrightarrow P \vdash u @ w \Rightarrow(n)\ v @ w$
apply (*induction n arbitrary*: *v*)
apply *simp*
using *derive_append* **by** (*fastforce simp*: *relpowp_Suc_right*)

lemma *deriven_prepend*:
P ⊢ *u* ⇒(*n*) *v* ⟹ *P* ⊢ *w* @ *u* ⇒(*n*) *w* @ *v*
apply (*induction n arbitrary*: *v*)
apply *simp*
using *derive_prepend* **by** (*fastforce simp*: *relpowp_Suc_right*)

lemma *derives_append*:
P ⊢ *u* ⇒∗ *v* ⟹ *P* ⊢ *u*@*w* ⇒∗ *v*@*w*
by (*metis deriven_append rtranclp_power*)

lemma *derives_prepend*:
P ⊢ *u* ⇒∗ *v* ⟹ *P* ⊢ *w*@*u* ⇒∗ *w*@*v*
by (*metis deriven_prepend rtranclp_power*)

lemma *derive_append_decomp*:
P ⊢ *u*@*v* ⇒ *w* ⟷
(∃ *u*′. *w* = *u*′@*v* ∧ *P* ⊢ *u* ⇒ *u*′) ∨ (∃ *v*′. *w* = *u*@*v*′ ∧ *P* ⊢ *v* ⇒ *v*′)
(**is** *?l* ⟷ *?r*)
proof
assume *?l*
then obtain *A r u1 u2*
where *Ar*: (*A*,*r*) ∈ *P*
and *uv*: *u*@*v* = *u1* @ *Nt A* # *u2*
and *w*: *w* = *u1* @ *r* @ *u2*
by (*auto simp*: *derive_iff*)
from *uv* **have** (∃ *s*. *u2* = *s* @ *v* ∧ *u* = *u1* @ *Nt A* # *s*) ∨
(∃ *s*. *u1* = *u*@*s* ∧ *v* = *s* @ *Nt A* # *u2*)
by (*auto simp*: *append_eq_append_conv2 append_eq_Cons_conv*)
with *Ar w* **show** *?r* **by** (*fastforce simp*: *derive_iff*)
next
show *?r* ⟹ *?l*
by (*auto simp add*: *derive_append derive_prepend*)
qed

lemma *deriven_append_decomp*:
P ⊢ *u* @ *v* ⇒(*n*) *w* ⟷
(∃ *n1 n2 w1 w2*. *n* = *n1* + *n2* ∧ *w* = *w1* @ *w2* ∧ *P* ⊢ *u* ⇒(*n1*) *w1* ∧ *P* ⊢ *v*
⇒(*n2*) *w2*)
(**is** *?l* ⟷ *?r*)
proof
show *?l* ⟹ *?r*
proof (*induction n arbitrary*: *u v*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
from *Suc.prems*
obtain *u*′ *v*′

```
    where or: P ⊢ u ⇒ u' ∧ v' = v ∨ u' = u ∧ P ⊢ v ⇒ v'
      and n: P ⊢ u'@v' ⇒(n) w
    by (fastforce simp: relpowp_Suc_left derive_append_decomp)
  from Suc.IH[OF n] or
  show ?case
    apply (elim disjE)
     apply (metis add_Suc relpowp_Suc_I2)
    by (metis add_Suc_right relpowp_Suc_I2)
 qed
next
 assume ?r
 then obtain n1 n2 w1 w2
   where [simp]: n = n1 + n2 w = w1 @ w2
     and u: P ⊢ u ⇒(n1) w1 and v: P ⊢ v ⇒(n2) w2
   by auto
 from u deriven_append
 have P ⊢ u @ v ⇒(n1) w1 @ v by fastforce
 also from v deriven_prepend
 have P ⊢ w1 @ v ⇒(n2) w1 @ w2 by fastforce
 finally show ?l by auto
qed

lemma derives_append_decomp:
 P ⊢ u @ v ⇒* w ⟷ (∃ u' v'. P ⊢ u ⇒* u' ∧ P ⊢ v ⇒* v' ∧ w = u' @ v')
 by (auto simp: rtranclp_power deriven_append_decomp)

lemma derives_concat:
 ∀ i ∈ set is. P ⊢ f i ⇒* g i ⟹ P ⊢ concat(map f is) ⇒* concat(map g is)
proof(induction is)
 case Nil
 then show ?case by auto
next
 case Cons
 thus ?case by(auto simp: derives_append_decomp less_Suc_eq)
qed

lemma derives_concat1:
 ∀ i ∈ set is. P ⊢ [f i] ⇒* g i ⟹ P ⊢ map f is ⇒* concat(map g is)
using derives_concat[where f = λi. [f i]] by auto

lemma derive_Cons:
P ⊢ u ⇒ v ⟹ P ⊢ a#u ⇒ a#v
 using derive_prepend[of P u v [a]] by auto

lemma derives_Cons:
R ⊢ u ⇒* v ⟹ R ⊢ a#u ⇒* a#v
 using derives_prepend[of _ _ _ [a]] by simp

lemma derive_from_empty[simp]:
```

$P \vdash [] \Rightarrow w \longleftrightarrow \mathit{False}$
by (*auto simp*: *derive_iff*)

lemma *deriven_from_empty*[*simp*]:
$P \vdash [] \Rightarrow(n)\ w \longleftrightarrow n = 0 \wedge w = []$
by (*induct n*, *auto simp*: *relpowp_Suc_left*)

lemma *derives_from_empty*[*simp*]:
$\mathcal{G} \vdash [] \Rightarrow* w \longleftrightarrow w = []$
by (*auto elim*: *converse_rtranclpE*)

lemma *deriven_start1*:
assumes $P \vdash [Nt\ A] \Rightarrow(n)\ map\ Tm\ w$
shows $\exists \alpha\ m.\ n = Suc\ m \wedge P \vdash \alpha \Rightarrow(m)\ (map\ Tm\ w) \wedge (A,\alpha) \in P$
proof (*cases n*)
case *0*
thus *?thesis*
using *assms* **by** *auto*
next
case (*Suc m*)
then obtain α **where** $*$: $P \vdash [Nt\ A] \Rightarrow \alpha\ P \vdash \alpha \Rightarrow(m)\ map\ Tm\ w$
using *assms* **by** (*meson relpowp_Suc_E2*)
from *derive.cases*[*OF* $*(1)$] **have** $(A, \alpha) \in P$
by (*simp add*: *Cons_eq_append_conv*) *blast*
thus *?thesis* **using** $*(2)$ *Suc* **by** *auto*
qed

lemma *derives_start1*: $P \vdash [Nt\ A] \Rightarrow*\ map\ Tm\ w \implies \exists \alpha.\ P \vdash \alpha \Rightarrow*\ map\ Tm\ w \wedge (A,\alpha) \in P$
using *deriven_start1* **by** (*metis rtranclp_power*)

lemma *Lang_empty_if_notin_Lhss*: $A \notin Lhss\ P \implies Lang\ P\ A = \{\}$
unfolding *Lhss_def Lang_def*
using *derives_start1* **by** *fastforce*

lemma *derive_Tm_Cons*:
$P \vdash Tm\ a \#\ u \Rightarrow v \longleftrightarrow (\exists w.\ v = Tm\ a \#\ w \wedge P \vdash u \Rightarrow w)$
by (*fastforce simp*: *derive_iff Cons_eq_append_conv*)

lemma *deriven_Tm_Cons*:
$P \vdash Tm\ a \#\ u \Rightarrow(n)\ v \longleftrightarrow (\exists w.\ v = Tm\ a \#\ w \wedge P \vdash u \Rightarrow(n)\ w)$
proof (*induction n arbitrary*: *u*)
case *0*
show *?case* **by** *auto*
next
case (*Suc n*)
then show *?case* **by** (*force simp*: *derive_Tm_Cons relpowp_Suc_left OO_def*)
qed

lemma *deriven_Tms_prepend*: $R \vdash map\ Tm\ t\ @\ u \Rightarrow(n)\ v \Longrightarrow \exists\, v1.\ v = map\ Tm\ t\ @\ v1 \wedge R \vdash u \Rightarrow(n)\ v1$
by (*induction t arbitrary*: *v*) (*auto simp add*: *deriven_Tm_Cons*)

lemma *derives_Tm_Cons*:
$P \vdash Tm\ a\ \#\ u \Rightarrow*\ v \longleftrightarrow (\exists\, w.\ v = Tm\ a\ \#\ w \wedge P \vdash u \Rightarrow*\ w)$
by (*metis deriven_Tm_Cons rtranclp_power*)

lemma *derives_Tm*[*simp*]: $P \vdash [Tm\ a] \Rightarrow*\ w \longleftrightarrow w = [Tm\ a]$
by(*simp add*: *derives_Tm_Cons*)

lemma *derive_singleton*: $P \vdash [a] \Rightarrow u \longleftrightarrow (\exists\, A.\ (A,u) \in P \wedge a = Nt\ A)$
by (*auto simp*: *derive_iff Cons_eq_append_conv*)

lemma *deriven_singleton*: $P \vdash [a] \Rightarrow(n)\ u \longleftrightarrow$ (
case n of 0 $\Rightarrow u = [a]$
| *Suc m* $\Rightarrow \exists (A,w) \in P.\ a = Nt\ A \wedge P \vdash w \Rightarrow(m)\ u$)
(**is** *?l* $\longleftrightarrow$ *?r*)
proof
show *?l* $\Longrightarrow$ *?r*
proof (*induction n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
then show *?case*
by (*smt* (*verit*, *ccfv_threshold*) *case_prod_conv derive_singleton nat.simps*(*5*) *relpowp_Suc_E2*)
qed
show *?r* $\Longrightarrow$ *?l*
by (*auto simp*: *derive_singleton relpowp_Suc_I2 split*: *nat.splits*)
qed

lemma *deriven_Cons_decomp*:
$P \vdash a\ \#\ u \Rightarrow(n)\ v \longleftrightarrow$
$(\exists\, v2.\ v = a\#v2 \wedge P \vdash u \Rightarrow(n)\ v2) \vee$
$(\exists\, n1\ n2\ A\ w\ v1\ v2.\ n = Suc\ (n1 + n2) \wedge v = v1\ @\ v2 \wedge a = Nt\ A \wedge$
$(A,w) \in P \wedge P \vdash w \Rightarrow(n1)\ v1 \wedge P \vdash u \Rightarrow(n2)\ v2)$
(**is** *?l* = *?r*)
proof
assume *?l*
then obtain *n1 n2 v1 v2*
where [*simp*]: $n = n1 + n2\ v = v1\ @\ v2$
and *1*: $P \vdash [a] \Rightarrow(n1)\ v1$ **and** *2*: $P \vdash u \Rightarrow(n2)\ v2$
unfolding *deriven_append_decomp*[*of n P* [*a*] *u v*,*simplified*]
by *auto*
show *?r*
proof (*cases n1*)
case *0*

```
    with 1 2 show ?thesis by auto
  next
    case [simp]: (Suc m)
    with 1 obtain A w
      where [simp]: a = Nt A (A,w) ∈ P and w: P ⊢ w ⇒(m) v1
      by (auto simp: deriven_singleton)
    with 2
    have n = Suc (m + n2) ∧ v = v1 @ v2 ∧ a = Nt A ∧
   (A,w) ∈ P ∧ P ⊢ w ⇒(m) v1 ∧ P ⊢ u ⇒(n2) v2
      by auto
    then show ?thesis
      by (auto simp: append_eq_Cons_conv)
  qed
next
  assume ?r
  then
  show ?l
  proof (elim disjE exE conjE)
    fix v2
    assume [simp]: v = a # v2 and u: P ⊢ u ⇒(n) v2
    from deriven_prepend[OF u, of [a]]
    show ?thesis
      by auto
  next
    fix n1 n2 A w v1 v2
    assume [simp]: n = Suc (n1 + n2) v = v1 @ v2 a = Nt A
      and Aw: (A, w) ∈ P
      and w: P ⊢ w ⇒(n1) v1
      and u: P ⊢ u ⇒(n2) v2
    have P ⊢ [a] ⇒ w
      by (simp add: Aw derive_singleton)
    with w have P ⊢ [a] ⇒(Suc n1) v1
      by (metis relpowp_Suc_I2)
    from deriven_append[OF this]
    have 1: P ⊢ a#u ⇒(Suc n1) v1@u
      by auto
    also have P ⊢ ... ⇒(n2) v1@v2
      using deriven_prepend[OF u].
    finally
    show ?thesis by simp
  qed
qed

lemma derives_Cons_decomp:
  P ⊢ s # u ⇒* v ⟷
  (∃ v2. v = s#v2 ∧ P ⊢ u ⇒* v2) ∨
  (∃ A w v1 v2. v = v1 @ v2 ∧ s = Nt A ∧ (A,w) ∈ P ∧ P ⊢ w ⇒* v1 ∧ P ⊢ u
⇒* v2) (is ?L ⟷ ?R)
proof
```

```
  assume ?L thus ?R using deriven_Cons_decomp[of _ P s u v] by (metis
rtranclp_power)
next
 assume ?R thus ?L by (meson derives_Cons derives_Cons_rule derives_append_decomp)
qed

lemma deriven_Suc_decomp_left:
  P ⊢ u ⇒(Suc n) v ⟷ (∃ p A u2 w v1 v2 n1 n2.
  u = p @ Nt A # u2 ∧ v = p @ v1 @ v2 ∧ n = n1 + n2 ∧
  (A,w) ∈ P ∧ P ⊢ w ⇒(n1) v1 ∧
  P ⊢ u2 ⇒(n2) v2) (is ?l ⟷ ?r)
proof
  show ?r ⟹ ?l
    by (auto intro!: deriven_prepend simp: deriven_Cons_decomp)
  show ?l ⟹ ?r
  proof(induction u arbitrary: v n)
    case Nil
    then show ?case by auto
  next
    case (Cons a u)
    from Cons.prems[unfolded deriven_Cons_decomp]
    show ?case
    proof (elim disjE exE conjE, goal_cases)
      case (1 v2)
      with Cons.IH[OF this(2)] show ?thesis
        by (metis append_Cons)
    next
      case (2 n1 n2 A w v1 v2)
      then show ?thesis by (fastforce simp:Cons_eq_append_conv)
    qed
  qed
qed

lemma derives_NilD: P ⊢ w ⇒* [] ⟹ s ∈ set w ⟹ P ⊢ [s] ⇒* []
proof(induction arbitrary: s rule: converse_derives_induct)
  case base
  then show ?case by simp
next
  case (step u A v w)
  then show ?case using derives_append_decomp[where u=[Nt A] and v=v]
    by (auto simp: derives_append_decomp)
qed

lemma derive_decomp_Tm: P ⊢ α ⇒(n) map Tm β ⟹
  ∃ βs ns. β = concat βs ∧ length α = length βs ∧ length α = length ns ∧ sum_list
ns = n
        ∧ (∀ i < length βs. P ⊢ [α ! i] ⇒(ns!i) map Tm (βs ! i))
  (is _ ⟹ ∃ βs ns. ?G α β n βs ns)
proof (induction α arbitrary: β n)
```

```
  case (Cons s α)
  from deriven_Cons_decomp[THEN iffD1, OF Cons.prems]
  show ?case
  proof (elim disjE exE conjE)
    fix γ assume as: map Tm β = s # γ P ⊢ α ⇒(n) γ
    then obtain s' γ' where β = s' # γ'  P ⊢ α ⇒(n) map Tm γ' s = Tm s' by
force
    from Cons.IH[OF this(2)] obtain βs ns where ∗: ?G α γ' n βs ns
      by blast
    let ?βs = [s']#βs
    let ?ns = 0#ns
    have ?G (s#α) β n ?βs ?ns
      using ‹β = _› as ∗ by (auto simp: nth_Cons')
    then show ?thesis by blast
  next
    fix n1 n2 A w β1 β2
    assume ∗: n = Suc (n1 + n2) map Tm β = β1 @ β2 s = Nt A (A, w) ∈ P
P ⊢ w ⇒(n1) β1 P ⊢ α ⇒(n2) β2
    then obtain β1' β2' where ∗∗: β = β1' @ β2' P ⊢ w ⇒(n1) map Tm β1'
P ⊢ α ⇒(n2) map Tm β2'
      by (metis (no_types, lifting) append_eq_map_conv)
    from Cons.IH[OF this(3)] obtain βs ns
      where ∗∗∗: ?G α β2' n2 βs ns
      by blast
    let ?βs = β1'#βs
    let ?ns = Suc n1 # ns
    from ∗ ∗∗ have P ⊢ [(s#α) ! 0] ⇒(?ns ! 0) map Tm (?βs ! 0)
      by (metis derive_singleton nth_Cons_0 relpowp_Suc_I2)
    then have ?G (s#α) β n ?βs ?ns
    using ∗ ∗∗ ∗∗∗ by (auto simp add: nth_Cons' derives_Cons_rule fold_plus_sum_list_rev)
    then show ?thesis by blast
  qed
qed simp

lemma word_decomp1:
  R ⊢ p @ [Nt A] @ map Tm ts ⇒(n) map Tm q
  ⟹ ∃ pt At w k m. R ⊢ p ⇒(k) map Tm pt ∧ R ⊢ w ⇒(m) map Tm At ∧ (A,
w) ∈ R
      ∧ q = pt @ At @ ts ∧ n = Suc(k + m)
proof −
  assume assm: R ⊢ p @ [Nt A] @ map Tm ts ⇒(n) map Tm q
  then obtain q1 where P: R ⊢ p@[Nt A] ⇒(n) q1 ∧ map Tm q = q1 @ map
Tm ts
    unfolding deriven_append_decomp
   by (metis add.commute add_0 append.assoc not_derive_from_Tms relpowp_E2)
  then obtain q1t where q1 = map Tm q1t q = q1t @ ts
    by (metis map_Tm_inject_iff map_eq_append_conv)
  with P obtain pt At w k m where P2: R ⊢ p ⇒(k) map Tm pt ∧ R ⊢ w ⇒(m)
map Tm At ∧ (A, w) ∈ R
```

```
    ∧ q1t = pt @ At ∧ n = Suc(k + m)
    by(fastforce simp: deriven_append_decomp map_eq_append_conv dest: deriven_start1)
  then have q = pt @ At @ ts using ‹q = _› by simp
  then show ?thesis using P2 by blast
qed

lemma deriven_start_sent:
  R ⊢ u @ Nt V # w ⇒(Suc n) map Tm x ⟹ ∃ v. (V, v) ∈ R ∧ R ⊢ u @ v @ w ⇒(n) map Tm x
proof −
  assume assm: R ⊢ u @ Nt V # w ⇒(Suc n) map Tm x
  then obtain n1 n2 xu xvw
    where P1: Suc n = n1 + n2 ∧ map Tm x = xu @ xvw ∧ R ⊢ u ⇒(n1) xu ∧ R ⊢ Nt V # w ⇒(n2) xvw
    by (auto simp add: deriven_append_decomp)
  then have t: ∄ t. xvw = Nt V # t
    by (metis append_eq_map_conv map_eq_Cons_D sym.distinct(1))
  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
    using P1 t by (auto simp add: deriven_Cons_decomp)
  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

lemma derives_simul_rules:
  assumes ⋀A w. (A,w) ∈ P ⟹ P' ⊢ [Nt A] ⇒* w
  shows P ⊢ w ⇒* w' ⟹ P' ⊢ w ⇒* w'
proof(induction rule: derives_induct)
  case base
  then show ?case by simp
next
  case (step u A v w)
  then show ?case
    by (meson assms derives_append derives_prepend rtranclp_trans)
qed

lemma derives_set_subset:
  P ⊢ u ⇒* v ⟹ set v ⊆ set u ∪ Syms P
proof (induction rule: derives_induct)
  case base
  then show ?case by simp
```

next
  case (*step u A v w*)
  then show *?case* **unfolding** *Syms_def* **by** (*auto*)
qed

lemma *derives_nts_syms_subset*:
  $P \vdash u \Rightarrow* v \Longrightarrow nts\_syms\ v \subseteq nts\_syms\ u \cup Nts\ P$
by(*drule derives_set_subset*)
  (*auto simp*: *nts_syms_def Nts_def Syms_def*)

lemma *derives_tms_syms_subset*:
  $P \vdash u \Rightarrow* v \Longrightarrow tms\_syms\ v \subseteq tms\_syms\ u \cup Tms\ P$
by(*drule derives_set_subset*)
  (*auto simp*: *tms_syms_def Tms_def Syms_def*)

Bottom-up definition of $\Rightarrow*$. Single definition yields more compact inductions. But *derives_induct* may already do the job.

inductive *derives_bu* :: ('*n*, '*t*) *Prods* ⇒ ('*n*,'*t*) *syms* ⇒ ('*n*,'*t*) *syms* ⇒ *bool*
  ((*2*_ ⊢/ (_/ ⇒*bu*/ _)) [*50*, *0*, *50*] *50*) **for** *P* :: ('*n*, '*t*) *Prods*
  where
bu_refl: $P \vdash \alpha \Rightarrow bu\ \alpha$ |
bu_prod: $(A,\alpha) \in P \Longrightarrow P \vdash [Nt\ A] \Rightarrow bu\ \alpha$ |
bu_embed: $[\![\ P \vdash \alpha \Rightarrow bu\ \alpha_1\ @\ \alpha_2\ @\ \alpha_3;\ P \vdash \alpha_2 \Rightarrow bu\ \beta\ ]\!] \Longrightarrow P \vdash \alpha \Rightarrow bu\ \alpha_1\ @\ \beta\ @\ \alpha_3$

lemma *derives_if_bu*: $P \vdash \alpha \Rightarrow bu\ \beta \Longrightarrow P \vdash \alpha \Rightarrow* \beta$
proof(*induction rule*: *derives_bu.induct*)
  case (*bu_refl*) **then show** *?case* **by** *simp*
next
  case (*bu_prod A α*) **then show** *?case* **by** (*simp add*: *derives_Cons_rule*)
next
  case (*bu_embed α α₁ α₂ α₃ β*) **then show** *?case*
    by (*meson derives_append derives_prepend rtranclp_trans*)
qed

lemma *derives_bu_if*: $P \vdash \alpha \Rightarrow* \beta \Longrightarrow P \vdash \alpha \Rightarrow bu\ \beta$
proof(*induction rule*: *derives_induct*)
  case *base*
  then show *?case* **by** (*simp add*: *bu_refl*)
next
  case (*step u A v w*)
  then show *?case*
    by (*meson bu_embed bu_prod*)
qed

lemma *derives_bu_iff*: $P \vdash \alpha \Rightarrow bu\ \beta \longleftrightarrow P \vdash \alpha \Rightarrow* \beta$
by (*meson derives_bu_if derives_if_bu*)

1.2.4 Leftmost/Rightmost Derivations

inductive *derivel* :: ('n,'t) *Prods* ⇒ ('n,'t) *syms* ⇒ ('n,'t)*syms* ⇒ *bool*
((2_ ⊢/ (_ ⇒*l*/ _)) [50, 0, 50] 50) **where**
(*A*,*α*) ∈ *P* ⟹ *P* ⊢ *map Tm u* @ *Nt A* # *v* ⇒*l map Tm u* @ *α* @ *v*

abbreviation *derivels* ((2_ ⊢/ (_ ⇒*l*∗/ _)) [50, 0, 50] 50) **where**
P ⊢ *u* ⇒*l*∗ *v* ≡ ((*derivel P*) ^**) *u v*

abbreviation *derivels1* ((2_ ⊢/ (_ ⇒*l*+/ _)) [50, 0, 50] 50) **where**
P ⊢ *u* ⇒*l*+ *v* ≡ ((*derivel P*) ^++) *u v*

abbreviation *deriveln* ((2_ ⊢/ (_ ⇒*l*'(_')/ _)) [50, 0, 0, 50] 50) **where**
P ⊢ *u* ⇒*l*(*n*) *v* ≡ ((*derivel P*) ^^ *n*) *u v*

inductive *deriver* :: ('n,'t) *Prods* ⇒ ('n,'t) *syms* ⇒ ('n,'t)*syms* ⇒ *bool*
((2_ ⊢/ (_ ⇒*r*/ _)) [50, 0, 50] 50) **where**
(*A*,*α*) ∈ *P* ⟹ *P* ⊢ *u* @ *Nt A* # *map Tm v* ⇒*r u* @ *α* @ *map Tm v*

abbreviation *derivers* ((2_ ⊢/ (_ ⇒*r*∗/ _)) [50, 0, 50] 50) **where**
P ⊢ *u* ⇒*r*∗ *v* ≡ ((*deriver P*) ^**) *u v*

abbreviation *derivers1* ((2_ ⊢/ (_ ⇒*r*+/ _)) [50, 0, 50] 50) **where**
P ⊢ *u* ⇒*r*+ *v* ≡ ((*deriver P*) ^++) *u v*

abbreviation *derivern* ((2_ ⊢/ (_ ⇒*r*'(_')/ _)) [50, 0, 0, 50] 50) **where**
P ⊢ *u* ⇒*r*(*n*) *v* ≡ ((*deriver P*) ^^ *n*) *u v*

lemma *derivel_iff*: *R* ⊢ *u* ⇒*l v* ⟷
(∃ (*A*,*w*) ∈ *R*. ∃ *u1 u2*. *u* = *map Tm u1* @ *Nt A* # *u2* ∧ *v* = *map Tm u1* @ *w* @ *u2*)
by (*auto simp*: *derivel.simps*)

lemma *derivel_from_empty*[*simp*]:
P ⊢ [] ⇒*l w* ⟷ *False* **by** (*auto simp*: *derivel_iff*)

lemma *deriveln_from_empty*[*simp*]:
P ⊢ [] ⇒*l*(*n*) *w* ⟷ *n* = *0* ∧ *w* = []
by (*induct n*, *auto simp*: *relpowp_Suc_left*)

lemma *derivels_from_empty*[*simp*]:
G ⊢ [] ⇒*l*∗ *w* ⟷ *w* = []
by (*auto elim*: *converse_rtranclpE*)

lemma *Un_derivel*: *R* ∪ *S* ⊢ *y* ⇒*l z* ⟷ *R* ⊢ *y* ⇒*l z* ∨ *S* ⊢ *y* ⇒*l z*
by (*fastforce simp*: *derivel_iff*)

lemma *derivel_append*:
P ⊢ *u* ⇒*l v* ⟹ *P* ⊢ *u* @ *w* ⇒*l v* @ *w*

by (*force simp*: *derivel_iff*)

lemma *deriveln_append*:
$P \vdash u \Rightarrow l(n)\ v \Longrightarrow P \vdash u\ @\ w \Rightarrow l(n)\ v\ @\ w$
proof (*induction n arbitrary*: *u*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
then obtain *y* **where** *uy*: $P \vdash u \Rightarrow l\ y$ **and** *yv*: $P \vdash y \Rightarrow l(n)\ v$
by (*auto simp*: *relpowp_Suc_left*)
have $P \vdash u\ @\ w \Rightarrow l\ y\ @\ w$
using *derivel_append*[*OF uy*]**.**
also from *Suc.IH yv* **have** $P \vdash \ldots \Rightarrow l(n)\ v\ @\ w$ **by** *auto*
finally show *?case***.**
qed

lemma *derivels_append*:
$P \vdash u \Rightarrow l*\ v \Longrightarrow P \vdash u\ @\ w \Rightarrow l*\ v\ @\ w$
by (*metis deriveln_append rtranclp_power*)

lemma *derivels1_append*:
$P \vdash u \Rightarrow l+\ v \Longrightarrow P \vdash u\ @\ w \Rightarrow l+\ v\ @\ w$
by (*metis deriveln_append tranclp_power*)

lemma *derivel_Tm_Cons*:
$P \vdash Tm\ a\ \#\ u \Rightarrow l\ v \longleftrightarrow (\exists\, w.\ v = Tm\ a\ \#\ w \wedge P \vdash u \Rightarrow l\ w)$
apply (*cases v*)
apply (*simp add*: *derivel_iff*)
apply (*simp simp*: *derivel.simps Cons_eq_append_conv Cons_eq_map_conv*)
done

lemma *deriveln_Tm_Cons*:
$P \vdash Tm\ a\ \#\ u \Rightarrow l(n)\ v \longleftrightarrow (\exists\, w.\ v = Tm\ a\ \#\ w \wedge P \vdash u \Rightarrow l(n)\ w)$
by (*induction n arbitrary*: *u v*;
fastforce simp: *derivel_Tm_Cons relpowp_Suc_right OO_def*)

lemma *derivels_Tm_Cons*:
$P \vdash Tm\ a\ \#\ u \Rightarrow l*\ v \longleftrightarrow (\exists\, w.\ v = Tm\ a\ \#\ w \wedge P \vdash u \Rightarrow l*\ w)$
by (*metis deriveln_Tm_Cons rtranclp_power*)

lemma *derivel_Nt_Cons*:
$P \vdash Nt\ A\ \#\ u \Rightarrow l\ v \longleftrightarrow (\exists\, w.\ (A,w) \in P \wedge v = w\ @\ u)$
by (*auto simp*: *derivel_iff Cons_eq_append_conv Cons_eq_map_conv*)

lemma *derivels1_Nt_Cons*:
$P \vdash Nt\ A\ \#\ u \Rightarrow l+\ v \longleftrightarrow (\exists\, w.\ (A,w) \in P \wedge P \vdash w\ @\ u \Rightarrow l*\ v)$
by (*auto simp*: *tranclp_unfold_left derivel_Nt_Cons OO_def*)

lemma *derivels_Nt_Cons*:
  $P \vdash Nt\ A\ \#\ u \Rightarrow l*\ v \longleftrightarrow v = Nt\ A\ \#\ u \vee (\exists\, w.\ (A,w) \in P \wedge P \vdash w\ @\ u \Rightarrow l*\ v)$
  by (*auto simp*: *Nitpick.rtranclp_unfold derivels1_Nt_Cons*)

lemma *deriveln_Nt_Cons*:
  $P \vdash Nt\ A\ \#\ u \Rightarrow l(n)\ v \longleftrightarrow$ (
  case n of 0 $\Rightarrow v = Nt\ A\ \#\ u$
  | *Suc m* $\Rightarrow \exists\, w.\ (A,w) \in P \wedge P \vdash w\ @\ u \Rightarrow l(m)\ v$)
  by (*cases n*) (*auto simp*: *derivel_Nt_Cons relpowp_Suc_left OO_def*)

lemma *derivel_map_Tm_append*:
  $P \vdash map\ Tm\ w\ @\ u \Rightarrow l\ v \longleftrightarrow (\exists\, x.\ v = map\ Tm\ w\ @\ x \wedge P \vdash u \Rightarrow l\ x)$
  apply (*induction w arbitrary*:*v*)
  by (*auto simp*: *derivel_Tm_Cons Cons_eq_append_conv*)

lemma *deriveln_map_Tm_append*:
  $P \vdash map\ Tm\ w\ @\ u \Rightarrow l(n)\ v \longleftrightarrow (\exists\, x.\ v = map\ Tm\ w\ @\ x \wedge P \vdash u \Rightarrow l(n)\ x)$
  by (*induction n arbitrary*: *u*;
      force simp: *derivel_map_Tm_append relpowp_Suc_left OO_def*)

lemma *derivels_map_Tm_append*:
  $P \vdash map\ Tm\ w\ @\ u \Rightarrow l*\ v \longleftrightarrow (\exists\, x.\ v = map\ Tm\ w\ @\ x \wedge P \vdash u \Rightarrow l*\ x)$
  by (*metis deriveln_map_Tm_append rtranclp_power*)

lemma *derivel_not_elim_Tm*:
  assumes $P \vdash xs \Rightarrow l\ map\ Nt\ w$
  shows $\exists\, v.\ xs = map\ Nt\ v$
proof −
  from *assms* **obtain** $A\ \alpha\ u\ xs'$ **where**
        A_w: $(A,\ \alpha) \in P$
      and *xs*: $xs = map\ Tm\ u\ @\ Nt\ A\ \#\ xs'$
      and *ys*: $map\ Nt\ w = map\ Tm\ u\ @\ \alpha\ @\ xs'$
    unfolding *derivel_iff* **by** *fast*

  from *ys* **have** *u1*: $u = []$
    by (*metis Nil_is_append_conv Nil_is_map_conv hd_append list.map_sel*(*1*) *sym.simps*(*4*))
  moreover from *ys* **obtain** u' **where** $xs' = map\ Nt\ u'$
    by (*metis append_eq_map_conv*)

  ultimately have $xs = map\ Nt\ (A\ \#\ u')$
    by (*simp add*: *xs*)
  then show *?thesis* **by** *blast*
qed

lemma *deriveln_not_elim_Tm*:
  assumes $P \vdash xs \Rightarrow l(n)\ map\ Nt\ w$
  shows $\exists\, v.\ xs = map\ Nt\ v$

using *assms*
proof (*induction n arbitrary*: *xs*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc n*)
then obtain *z* **where** $P \vdash xs \Rightarrow l\ z$ **and** $P \vdash z \Rightarrow l(n)\ map\ Nt\ w$
using *relpowp_Suc_E2* **by** *metis*
with *Suc* **show** *?case* **using** *derivel_not_elim_Tm*
by *blast*
qed

lemma *decomp_derivel_map_Nts*:
assumes $P \vdash map\ Nt\ Xs \Rightarrow l\ map\ Nt\ Zs$
shows $\exists X\ Xs'\ Ys.\ Xs = X\ \#\ Xs' \wedge P \vdash [Nt\ X] \Rightarrow l\ map\ Nt\ Ys \wedge Zs = Ys\ @\ Xs'$
using *assms* **unfolding** *derivel_iff*
by (*fastforce simp*: *map_eq_append_conv split*: *prod.splits*)

lemma *derivel_imp_derive*: $P \vdash u \Rightarrow l\ v \Longrightarrow P \vdash u \Rightarrow v$
using *derive.simps derivel.cases self_append_conv2* **by** *fastforce*

lemma *deriveln_imp_deriven*:
$P \vdash u \Rightarrow l(n)\ v \Longrightarrow P \vdash u \Rightarrow (n)\ v$
using *relpowp_mono derivel_imp_derive* **by** *metis*

lemma *derivels_imp_derives*:
$P \vdash u \Rightarrow l*\ v \Longrightarrow P \vdash u \Rightarrow *\ v$
by (*metis derivel_imp_derive mono_rtranclp*)

lemma *deriveln_iff_deriven*:
$P \vdash u \Rightarrow l(n)\ map\ Tm\ v \longleftrightarrow P \vdash u \Rightarrow (n)\ map\ Tm\ v$
(**is** *?l* $\longleftrightarrow$ *?r*)
proof
show *?l* $\Longrightarrow$ *?r* **using** *deriveln_imp_deriven*.
assume *?r*
then show *?l*
proof (*induction n arbitrary*: *u v rule*: *less_induct*)
case (*less n*)
from ‹$P \vdash u \Rightarrow (n)\ map\ Tm\ v$›
show *?case*
proof (*induction u arbitrary*: *v*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a u*)
show *?case*
using *Cons.prems*(*1*) *Cons.IH less.IH*
by (*auto simp*: *deriven_Cons_decomp deriveln_Tm_Cons deriveln_Nt_Cons*)

```
        (metis deriven_append_decomp lessI)
    qed
  qed
qed

lemma derivels_iff_derives: P ⊢ u ⇒l* map Tm v ⟷ P ⊢ u ⇒* map Tm v
  using deriveln_iff_deriven
  by (metis rtranclp_power)

lemma deriver_iff: R ⊢ u ⇒r v ⟷
  (∃ (A,w) ∈ R. ∃ u1 u2. u = u1 @ Nt A # map Tm u2 ∧ v = u1 @ w @ map
Tm u2)
  by (auto simp: deriver.simps)

lemma deriver_imp_derive: R ⊢ u ⇒r v ⟹ R ⊢ u ⇒ v
  by (auto simp: deriver_iff derive_iff)

lemma derivern_imp_deriven: R ⊢ u ⇒r(n) v ⟹ R ⊢ u ⇒(n) v
  by (metis (no_types, lifting) deriver_imp_derive relpowp_mono)

lemma derivers_imp_derives: R ⊢ u ⇒r* v ⟹ R ⊢ u ⇒* v
  by (metis deriver_imp_derive mono_rtranclp)

lemma deriver_iff_rev_derivel:
  P ⊢ u ⇒r v ⟷ map_prod id rev ‘ P ⊢ rev u ⇒l rev v (is ?l ⟷ ?r)
proof
  assume ?l
  then obtain A w u1 u2 where Aw: (A,w) ∈ P
    and u: u = u1 @ Nt A # map Tm u2
    and v: v = u1 @ w @ map Tm u2 by (auto simp: deriver.simps)
  from bexI[OF _ Aw] have (A, rev w) ∈ map_prod id rev ‘ P by (auto simp:
image_def)
  from derivel.intros[OF this, of rev u2 rev u1] u v
  show ?r by (simp add: rev_map)
next
  assume ?r
  then obtain A w u1 u2 where Aw: (A,w) ∈ P
    and u: u = u1 @ Nt A # map Tm u2
    and v: v = u1 @ w @ map Tm u2
    by (auto simp: derivel_iff rev_eq_append_conv rev_map)
  then show ?l by (auto simp: deriver_iff)
qed

lemma rev_deriver_iff_derivel:
  map_prod id rev ‘ P ⊢ u ⇒r v ⟷ P ⊢ rev u ⇒l rev v
  by (simp add: deriver_iff_rev_derivel image_image prod.map_comp o_def)

lemma derivern_iff_rev_deriveln:
  P ⊢ u ⇒r(n) v ⟷ map_prod id rev ‘ P ⊢ rev u ⇒l(n) rev v
```

proof (*induction n arbitrary*: *u*)
case *0*
show *?case* **by** *simp*
next
case (*Suc n*)
show *?case*
apply (*unfold relpowp_Suc_left OO_def*)
apply (*unfold Suc deriver_iff_rev_derivel*)
by (*metis rev_rev_ident*)
qed

lemma *rev_derivern_iff_deriveln*:
map_prod id rev ‘ P ⊢ u ⇒r(n) v ⟷ P ⊢ rev u ⇒l(n) rev v
by (*simp add*: *derivern_iff_rev_deriveln image_image prod.map_comp o_def*)

lemma *derivers_iff_rev_derivels*:
P ⊢ u ⇒r∗ v ⟷ map_prod id rev ‘ P ⊢ rev u ⇒l∗ rev v
using *derivern_iff_rev_deriveln*
by (*metis rtranclp_power*)

lemma *rev_derivers_iff_derivels*:
map_prod id rev ‘ P ⊢ u ⇒r∗ v ⟷ P ⊢ rev u ⇒l∗ rev v
by (*simp add*: *derivers_iff_rev_derivels image_image prod.map_comp o_def*)

lemma *rev_derive*:
map_prod id rev ‘ P ⊢ u ⇒ v ⟷ P ⊢ rev u ⇒ rev v
by (*force simp*: *derive_iff rev_eq_append_conv bex_pair_conv in_image_map_prod*
intro: *exI*[*of _ rev _*])

lemma *rev_deriven*:
map_prod id rev ‘ P ⊢ u ⇒(n) v ⟷ P ⊢ rev u ⇒(n) rev v
apply (*induction n arbitrary*: *u*)
apply *simp*
by (*auto simp*: *relpowp_Suc_left OO_def rev_derive intro*: *exI*[*of _ rev _*])

lemma *rev_derives*:
map_prod id rev ‘ P ⊢ u ⇒∗ v ⟷ P ⊢ rev u ⇒∗ rev v
using *rev_deriven*
by (*metis rtranclp_power*)

lemma *derivern_iff_deriven*: *P ⊢ u ⇒r(n) map Tm v ⟷ P ⊢ u ⇒(n) map Tm*
v
by (*auto simp*: *derivern_iff_rev_deriveln rev_map deriveln_iff_deriven rev_deriven*)

lemma *derivers_iff_derives*: *P ⊢ u ⇒r∗ map Tm v ⟷ P ⊢ u ⇒∗ map Tm v*
by (*simp add*: *derivern_iff_deriven rtranclp_power*)

lemma *derivern_prepend*: *R ⊢ u ⇒r(n) v ⟹ R ⊢ p @ u ⇒r(n) p @ v*
by (*fastforce simp*: *derivern_iff_rev_deriveln rev_map deriveln_append rev_eq_append_conv*)

lemma *deriver_append_map_Tm*:
P ⊢ *u* @ *map Tm w* ⇒*r v* ⟷ (∃ *x*. *v* = *x* @ *map Tm w* ∧ *P* ⊢ *u* ⇒*r x*)
by (*fastforce simp*: *deriver_iff_rev_derivel rev_map derivel_map_Tm_append rev_eq_append_conv*)

lemma *derivern_append_map_Tm*:
P ⊢ *u* @ *map Tm w* ⇒*r*(*n*) *v* ⟷ (∃ *x*. *v* = *x* @ *map Tm w* ∧ *P* ⊢ *u* ⇒*r*(*n*) *x*)
by (*fastforce simp*: *derivern_iff_rev_deriveln rev_map deriveln_map_Tm_append rev_eq_append_conv*)

lemma *deriver_snoc_Nt*:
P ⊢ *u* @ [*Nt A*] ⇒*r v* ⟷ (∃ *w*. (*A*,*w*) ∈ *P* ∧ *v* = *u* @ *w*)
by (*force simp*: *deriver_iff_rev_derivel derivel_Nt_Cons rev_eq_append_conv*)

lemma *deriver_singleton*:
P ⊢ [*Nt A*] ⇒*r v* ⟷ (*A*,*v*) ∈ *P*
using *deriver_snoc_Nt*[*of P* []] **by** *auto*

lemma *derivers1_snoc_Nt*:
P ⊢ *u* @ [*Nt A*] ⇒*r*+ *v* ⟷ (∃ *w*. (*A*,*w*) ∈ *P* ∧ *P* ⊢ *u* @ *w* ⇒*r*∗ *v*)
by (*auto simp*: *tranclp_unfold_left deriver_snoc_Nt OO_def*)

lemma *derivers_snoc_Nt*:
P ⊢ *u* @ [*Nt A*] ⇒*r*∗ *v* ⟷ *v* = *u* @ [*Nt A*] ∨ (∃ *w*. (*A*,*w*) ∈ *P* ∧ *P* ⊢ *u* @ *w* ⇒*r*∗ *v*)
by (*auto simp*: *Nitpick.rtranclp_unfold derivers1_snoc_Nt*)

lemma *derivern_snoc_Tm*:
P ⊢ *u* @ [*Tm a*] ⇒*r*(*n*) *v* ⟷ (∃ *w*. *v* = *w* @ [*Tm a*] ∧ *P* ⊢ *u* ⇒*r*(*n*) *w*)
by (*force simp*: *derivern_iff_rev_deriveln deriveln_Tm_Cons*)

lemma *derivern_snoc_Nt*:
P ⊢ *u* @ [*Nt A*] ⇒*r*(*n*) *v* ⟷ (
case n of 0 ⇒ *v* = *u* @ [*Nt A*]
| *Suc m* ⇒ ∃ *w*. (*A*,*w*) ∈ *P* ∧ *P* ⊢ *u* @ *w* ⇒*r*(*m*) *v*)
by (*cases n*) (*auto simp*: *relpowp_Suc_left deriver_snoc_Nt OO_def*)

lemma *derivern_singleton*:
P ⊢ [*Nt A*] ⇒*r*(*n*) *v* ⟷ (
case n of 0 ⇒ *v* = [*Nt A*]
| *Suc m* ⇒ ∃ *w*. (*A*,*w*) ∈ *P* ∧ *P* ⊢ *w* ⇒*r*(*m*) *v*)
using *derivern_snoc_Nt*[*of n P* [] *A v*] **by** (*cases n*, *auto*)

lemma *derivern_snoc_Nt_Tms_decomp1*:
R ⊢ *p* @ [*Nt A*] ⇒*r*(*n*) *map Tm q*
⟹ ∃ *pt At w k m*. *R* ⊢ *p* ⇒(*k*) *map Tm pt* ∧ *R* ⊢ *w* ⇒(*m*) *map Tm At* ∧ (*A*, *w*) ∈ *R*
∧ *q* = *pt* @ *At* ∧ *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

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

lemma *no_self_loops_derives*: $P \vdash u \Rightarrow(n)\ v \Longrightarrow \{p{\in}P.\ \neg(\exists\ A.\ p = (A,[Nt\ A]))\} \vdash u \Rightarrow* v$
proof(*induction n arbitrary*: *u*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
then have $\exists$ *w*. $P \vdash u \Rightarrow w \wedge P \vdash w \Rightarrow(n)\ v$
by (*simp add*: *relpowp_Suc_D2*)
then obtain *w* **where** *W*: $P \vdash u \Rightarrow w \wedge P \vdash w \Rightarrow(n)\ v$ **by** *blast*
then have $\exists$ (*A*,*x*) $\in$ *P*. $\exists$ *u1 u2*. *u* = *u1* @ *Nt A* # *u2* $\wedge$ *w* = *u1* @ *x* @ *u2*
by (*simp add*: *derive_iff*)
then obtain *A x u1 u2* **where** *prod*: *u* = *u1* @*Nt A*#*u2* $\wedge$ *w* = *u1* @*x*@*u2* $\wedge$ (*A*, *x*) $\in$ *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 $(A, x) \in \{p{\in}P.\ \neg(\exists\, A.\ p = (A,[Nt\ A]))\}$ **using** *prod* **by** (*auto*)
then show *?thesis* **using** *Suc W*
by (*metis* (*no_types*, *lifting*) *derives_rule prod rtranclp.rtrancl_refl*)
qed
qed

lemma *no_self_loops_derivels*: $P \vdash u \Rightarrow l(n)\ v \Longrightarrow \{p{\in}P.\ \neg(\exists\, A.\ p = (A,[Nt\ A]))\} \vdash u \Rightarrow l{*}\ v$
proof(*induction n arbitrary*: *u*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
then have $\exists\, w.\ P \vdash u \Rightarrow l\ w \wedge P \vdash w \Rightarrow l(n)\ v$
by (*simp add*: *relpowp_Suc_D2*)
then obtain *w* **where** *W*: $P \vdash u \Rightarrow l\ w \wedge P \vdash w \Rightarrow l(n)\ v$ **by** *blast*
then have $\exists\ (A,x) \in P.\ \exists\, u1\ u2.\ u = map\ Tm\ u1\ @\ Nt\ A\ \#\ u2 \wedge w = map\ Tm\ u1\ @\ x\ @\ u2$
by (*simp add*: *derivel_iff*)
then obtain *A x u1 u2* **where** *prod*: $u = map\ Tm\ u1\ @Nt\ A\#u2 \wedge w = map\ Tm\ u1\ @x@u2 \wedge (A,\ x) \in P$
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 $(A,\ x) \in \{p{\in}P.\ \neg(\exists\, A.\ p = (A,[Nt\ A]))\}$ **using** *prod* **by** (*auto*)
then show *?thesis* **using** *Suc W*
by (*metis* (*lifting*) *converse_rtranclp_into_rtranclp derivel.intros prod*)
qed
qed

1.3 Substitution in Lists

Function *substs y ys xs* replaces every occurrence of *y* in *xs* with the list *ys*

fun *substs* :: *'a* ⇒ *'a list* ⇒ *'a list* ⇒ *'a list* **where**
substs y ys [] = [] |
substs y ys (*x*#*xs*) = (*if x = y then ys* @ *substs y ys xs else x* # *substs y ys xs*)

Alternative definition, but apparently no simpler to use: *substs y ys xs* = *concat* (*map* (λ*x*. *if x = y then ys else* [*x*]) *xs*)

abbreviation *substsNt A* ≡ *substs* (*Nt A*)

lemma *substs_append*[*simp*]: *substs y ys* (*xs* @ *xs'*) = *substs y ys xs* @ *substs y ys xs'*
by (*induction xs*) *auto*

lemma *substs_skip*: $y \notin set\ xs \Longrightarrow substs\ y\ ys\ xs = xs$
by (*induction xs*) *auto*

lemma *susbstsNT_map_Tm*[*simp*]: $substsNt\ A\ \alpha\ (map\ Tm\ w) = map\ Tm\ w$
by(*rule substs_skip*) *auto*

lemma *substs_len*: $length\ (substs\ y\ [y']\ xs) = length\ xs$
by (*induction xs*) *auto*

lemma *substs_rev*: $y' \notin set\ xs \Longrightarrow substs\ y'\ [y]\ (substs\ y\ [y']\ xs) = xs$
by (*induction xs*) *auto*

lemma *substs_der*:
$(B,v) \in P \Longrightarrow P \vdash u \Rightarrow* substs\ (Nt\ B)\ v\ u$
proof (*induction u*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a u*)
then show *?case*
by (*auto simp add*: *derives_Cons_rule derives_prepend derives_Cons*)
qed

1.4 Epsilon-Freeness

definition *Eps_free* **where** $Eps\_free\ R = (\forall\,(\_,r) \in R.\ r \neq [])$

abbreviation *eps_free rs* == *Eps_free*(*set rs*)

lemma *Eps_freeI*:
assumes $\bigwedge A\ r.\ (A,r) \in R \Longrightarrow r \neq []$ **shows** *Eps_free R*
using *assms* **by** (*auto simp*: *Eps_free_def*)

lemma *Eps_free_Nil*: $Eps\_free\ R \Longrightarrow (A,[]) \notin R$
by (*auto simp*: *Eps_free_def*)

lemma *Eps_freeE_Cons*: $Eps\_free\ R \Longrightarrow (A,w) \in R \Longrightarrow \exists\, a\ u.\ w = a\#u$
by (*cases w*, *auto simp*: *Eps_free_def*)

lemma *Eps_free_derives_Nil*:
assumes *R*: *Eps_free R* **shows** $R \vdash l \Rightarrow* [] \longleftrightarrow l = []$ (**is** $?l \longleftrightarrow ?r$)
proof
show $?l \Longrightarrow ?r$
proof (*induction rule*: *converse_derives_induct*)
case *base*
show *?case* **by** *simp*
next
case (*step u A v w*)

then show *?case* **by** (*auto simp*: *Eps_free_Nil*[*OF R*])
qed
qed *auto*

lemma *Eps_free_deriven_Nil*:
⟦ *Eps_free R*; *R* ⊢ *l* ⇒(*n*) [] ⟧ ⟹ *l* = []
by (*metis Eps_free_derives_Nil relpowp_imp_rtranclp*)

lemma *Eps_free_derivels_Nil*: *Eps_free R* ⟹ *R* ⊢ *l* ⇒*l*∗ [] ⟷ *l* = []
by (*meson Eps_free_derives_Nil derivels_from_empty derivels_imp_derives*)

lemma *Eps_free_deriveln_Nil*: *Eps_free R* ⟹ *R* ⊢ *l* ⇒*l*(*n*) [] ⟹ *l* = []
by (*metis Eps_free_derivels_Nil relpowp_imp_rtranclp*)

lemma *decomp_deriveln_map_Nts*:
assumes *Eps_free P*
shows *P* ⊢ *Nt X* # *map Nt Xs* ⇒*l*(*n*) *map Nt Zs* ⟹
∃ *Ys*′. *Ys*′ @ *Xs* = *Zs* ∧ *P* ⊢ [*Nt X*] ⇒*l*(*n*) *map Nt Ys*′
proof (*induction n arbitrary*: *Zs*)
case *0*
then show *?case*
by (*auto*)
next
case (*Suc n*)
then obtain *ys* **where** *n*: *P* ⊢ *Nt X* # *map Nt Xs* ⇒*l*(*n*) *ys* **and** *P* ⊢ *ys* ⇒*l* *map Nt Zs*
using *relpowp_Suc_E* **by** *metis*
from ‹*P* ⊢ *ys* ⇒*l map Nt Zs*› **obtain** *Ys* **where** *ys* = *map Nt Ys*
using *derivel_not_elim_Tm* **by** *blast*
from *Suc.IH*[*of Ys*] *this n*
obtain *Ys*′ **where** *Ys* = *Ys*′ @ *Xs* ∧ *P* ⊢ [*Nt X*] ⇒*l*(*n*) *map Nt Ys*′ **by** *auto*
moreover from ‹*ys* = _› ‹*P* ⊢ *ys* ⇒*l map Nt Zs*› *decomp_derivel_map_Nts*[*of P Ys Zs*]
obtain *Y Xs*′ *Ysa* **where** *Ys* = *Y* # *Xs*′ ∧ *P* ⊢ [*Nt Y*] ⇒*l map Nt Ysa* ∧ *Zs* = *Ysa* @ *Xs*′ **by** *auto*
ultimately show *?case* **using** *Eps_free_deriveln_Nil*[*OF assms*, *of n* [*Nt X*]]
by (*auto simp*: *Cons_eq_append_conv derivel_Nt_Cons relpowp_Suc_I*)
qed

end

2 Parse Trees

theory *Parse_Tree*
imports *Context_Free_Grammar*
begin

datatype ('*n*,'*t*) *tree* = *Sym* ('*n*,'*t*) *sym* | *Rule* '*n* ('*n*,'*t*) *tree list*
datatype_compat *tree*

```
fun root :: ('n,'t) tree ⇒ ('n,'t) sym where
root(Sym s) = s |
root(Rule A _) = Nt A

fun fringe :: ('n,'t) tree ⇒ ('n,'t) syms where
fringe(Sym s) = [s] |
fringe(Rule _ ts) = concat(map fringe ts)

abbreviation fringes ts ≡ concat(map fringe ts)

fun parse_tree :: ('n,'t)Prods ⇒ ('n,'t) tree ⇒ bool where
parse_tree P (Sym s) = True |
parse_tree P (Rule A ts) = ((∀ t ∈ set ts. parse_tree P t) ∧ (A,map root ts) ∈ P)

lemma fringe_steps_if_parse_tree: parse_tree P t ⟹ P ⊢ [root t] ⇒* fringe t
proof(induction t)
  case (Sym s)
  then show ?case by (auto)
next
  case (Rule A ts)
  have P ⊢ [Nt A] ⇒ map root ts
    using Rule.prems by (simp add: derive_singleton)
  with Rule show ?case apply(simp)
    using derives_concat1 by (metis converse_rtranclp_into_rtranclp)
qed

fun subst_pt and subst_pts where
subst_pt t' 0 (Sym _) = t' |
subst_pt t' m (Rule A ts) = Rule A (subst_pts t' m ts) |
subst_pts t' m (t#ts) =
  (let n = length(fringe t) in if m < n then subst_pt t' m t # ts
   else t # subst_pts t' (m−n) ts)

lemma fringe_subst_pt: i < length(fringe t) ⟹
  fringe(subst_pt t' i t) = take i (fringe t) @ fringe t' @ drop (Suc i) (fringe t)
and
fringe_subst_pts: i < length(fringes ts) ⟹
  fringes (subst_pts t' i ts) =
    take i (fringes ts) @ fringe t' @ drop (Suc i) (fringes ts)
proof(induction t' i t and t' i ts rule: subst_pt_subst_pts.induct)
  case (3 t' m t ts)
  let ?n = length (fringe t)
  show ?case
  proof (cases m < ?n)
    case True
    thus ?thesis using 3.IH(1)[OF refl] by (simp add: Let_def)
  next
    case False
```

thus *?thesis* **using** *3.IH(2)*[*OF refl*] *3.prems* **by** (*simp add*: *Suc_diff_le*)
qed
qed *auto*

lemma *root_subst_pt*: ⟦ *i* < *length*(*fringe t*); *fringe t* ! *i* = *Nt A*; *root t′* = *Nt A* ⟧
⟹ *root* (*subst_pt t′ i t*) = *root t* **and**
map_root_subst_pts: ⟦ *i* < *length*(*fringes ts*); *fringes ts* ! *i* = *Nt A*; *root t′* = *Nt A* ⟧
⟹ *map root* (*subst_pts t′ i ts*) = *map root ts*
proof(*induction t′ i t* **and** *t′ i ts rule*: *subst_pt_subst_pts.induct*)
case (*3 t′ m t ts*)
then show *?case* **by** (*auto simp add*: *nth_append Let_def*)
qed *auto*

lemma *parse_tree_subst_pt*:
⟦ *parse_tree P t*; *i* < *length*(*fringe t*); *fringe t* ! *i* = *Nt A*; *parse_tree P t′*; *root t′*= *Nt A* ⟧
⟹ *parse_tree P* (*subst_pt t′ i t*)
and *parse_tree_subst_pts*:
⟦ ∀ *t* ∈ *set ts*. *parse_tree P t*; *i* < *length*(*fringes ts*); *fringes ts* ! *i* = *Nt A*; *parse_tree P t′*; *root t′*= *Nt A* ⟧
⟹ ∀ *t′* ∈ *set*(*subst_pts t′ i ts*). *parse_tree P t′*
proof(*induction t′ i t* **and** *t′ i ts rule*: *subst_pt_subst_pts.induct*)
case (*2 m A ts*)
then show *?case*
using *map_root_subst_pts* **by** *fastforce*
next
case (*3 m t ts*)
then show *?case* **by**(*auto simp add*: *nth_append Let_def*)
qed *auto*

lemma *parse_tree_if_derives*: *P* ⊢ [*Nt A*] ⇒∗ *w* ⟹ ∃ *t*. *parse_tree P t* ∧ *fringe t* = *w* ∧ *root t* = *Nt A*
proof(*induction rule*: *derives_induct*)
case *base*
then show *?case*
using *fringe.simps*(*1*) *parse_tree.simps*(*1*) *root.simps*(*1*) **by** *blast*
next
case (*step u A′ v w*)
then obtain *t* **where** *1*: *parse_tree P t* **and** *2*: *fringe t* = *u* @ [*Nt A′*] @ *v* **and** *3*: ‹*root t* = *Nt A*›
by *blast*
let *?t′* = *Rule A′* (*map Sym w*)
let *?t* = *subst_pt ?t′* (*length u*) *t*
have *fringe ?t* = *u* @ *w* @ *v*
using *2 fringe_subst_pt*[*of length u t ?t′*] **by**(*simp add*: *o_def*)
moreover have *parse_tree P ?t*
using *parse_tree_subst_pt*[*OF 1*, *of length u*] *step.hyps*(*2*) *2* **by**(*simp add*: *o_def*)

moreover have ‹*root ?t = Nt A*› **by** (*simp add*: *2 3 root_subst_pt*)
ultimately show *?case* **by** *blast*
qed

end

3 Renaming Nonterminals

theory *Renaming_CFG*
imports *Context_Free_Grammar*
begin

This theory provides lemmas that relate derivations w.r.t. some set of productions P to derivations w.r.t. a renaming of the nonterminals in P.

fun *rename_sym* :: (*'old* ⇒ *'new*) ⇒ (*'old*,*'t*) *sym* ⇒ (*'new*,*'t*) *sym* **where**
rename_sym f (*Nt n*) = *Nt* (*f n*) |
rename_sym f (*Tm t*) = *Tm t*

abbreviation *rename_syms f* ≡ *map* (*rename_sym f*)

fun *rename_prod* :: (*'old* ⇒ *'new*) ⇒ (*'old*,*'t*) *prod* ⇒ (*'new*,*'t*) *prod* **where**
rename_prod f (*A*,*w*) = (*f A*, *rename_syms f w*)

abbreviation *rename_Prods f P* ≡ *rename_prod f* ‘ *P*

lemma *rename_sym_o_Tm*[*simp*]: *rename_sym f* ∘ *Tm* = *Tm*
by(*rule ext*) *simp*

lemma *Nt_notin_rename_syms_if_notin_range*:
x ∉ *range f* ⟹ *Nt x* ∉ *set* (*rename_syms f w*)
by(*auto elim*!: *rename_sym.elims*[*OF sym*])

lemma *in_Nts_rename_Prods*: *B* ∈ *Nts* (*rename_Prods f P*) = (∃ *A* ∈ *Nts P*. *f A* = *B*)
unfolding *Nts_def nts_syms_def* **by**(*force split*: *prod.splits elim*!: *rename_sym.elims*[*OF sym*])

lemma *rename_preserves_deriven*:
P ⊢ *α* ⇒(*n*) *β* ⟹ *rename_Prods f P* ⊢ *rename_syms f α* ⇒(*n*) *rename_syms f β*
proof (*induction rule*: *deriven_induct*)
case *0*
then show *?case* **by** *simp*
next
let *?F* = *rename_syms f*
case (*Suc n u A v w*)
then have (*f A*, *rename_syms f w*) ∈ *rename_Prods f P*
by (*metis image_eqI rename_prod.simps*)

from *derive.intros*[*OF this*] **have** *rename_Prods f P* ⊢ *?F u @ ?F* [*Nt A*] *@ ?F v* ⇒ *?F u @ ?F w @ ?F v*
by *auto*
with *Suc* **show** *?case*
by (*simp add*: *relpowp_Suc_I*)
qed

lemma *rename_preserves_derives*:
P ⊢ *α* ⇒∗ *β* ⟹ *rename_Prods f P* ⊢ *rename_syms f α* ⇒∗ *rename_syms f β*
by (*meson rename_preserves_deriven rtranclp_power*)

lemma *rename_preserves_derivel*:
assumes *P* ⊢ *α* ⇒*l β*
shows *rename_Prods f P* ⊢ *rename_syms f α* ⇒*l rename_syms f β*
proof −
from *assms* **obtain** *A w u1 u2*
where *A_w_u1_u2*: (*A*,*w*) ∈ *P* ∧ *α* = *map Tm u1 @ Nt A # u2* ∧ *β* = *map Tm u1 @ w @ u2*
unfolding *derivel_iff* **by** *fast*
then have (*f A*, *rename_syms f w*) ∈ (*rename_Prods f P*) ∧
rename_syms f α = *map Tm u1 @ Nt* (*f A*) # *rename_syms f u2* ∧
rename_syms f β = *map Tm u1 @ rename_syms f w @ rename_syms f u2*
by *force*
then have ∃ (*A*,*w*) ∈ *rename_Prods f P*.
∃ *u1 u2*. *rename_syms f α* = *map Tm u1 @ Nt A # u2* ∧ *rename_syms f β* = *map Tm u1 @ w @ u2*
by *blast*
then show *?thesis* **by** (*simp only*: *derivel_iff*)
qed

lemma *rename_preserves_deriveln*:
P ⊢ *α* ⇒*l*(*n*) *β* ⟹ *rename_Prods f P* ⊢ *rename_syms f α* ⇒*l*(*n*) *rename_syms f β*
proof (*induction n arbitrary*: *β*)
case *0*
then show *?case* **by** *simp*
next
case *Suc* **then show** *?case*
by (*metis relpowp_Suc_E relpowp_Suc_I rename_preserves_derivel*)
qed

lemma *rename_preserves_derivels*:
P ⊢ *α* ⇒*l*∗ *β* ⟹ *rename_Prods f P* ⊢ *rename_syms f α* ⇒*l*∗ *rename_syms f β*
by (*meson rename_preserves_deriveln rtranclp_power*)

lemma *rename_deriven_iff_inj*:
fixes *P* :: (*'a*,*'t*)*Prods*

assumes *inj_on f* (*Nts P* ∪ *nts_syms α* ∪ *nts_syms β*)
shows *rename_Prods f P* ⊢ *rename_syms f α* ⇒(*n*) *rename_syms f β* ⟷ *P* ⊢ *α* ⇒(*n*) *β* (**is** *?l* ⟷ *?r*)
proof
show *?r* ⟹ *?l* **by** (*rule rename_preserves_deriven*)
next

let *?M* = *Nts P* ∪ *nts_syms α* ∪ *nts_syms β*
obtain *g* **where** *g* = *the_inv_into ?M f* **and** *inv*: (⋀*x*. *x* ∈ *?M* ⟹ (*g* (*f x*) = *x*))
using *assms* **by** (*simp add*: *the_inv_into_f_f inj_on_Un*)
then have *s* ∈ *Syms P* ∪ *set α* ∪ *set β* ⟹ *rename_sym g* (*rename_sym f s*) = *s* **for** *s*::('*a*,'*t*) *sym*
by (*cases s*) (*auto simp*: *Nts_def Syms_def nts_syms_def*)
then have *inv_rename_syms*: ⋀(*ss*::('*a*,'*t*) *syms*). *set ss* ⊆ *Syms P* ∪ *set α* ∪ *set β* ⟹ *rename_syms g* (*rename_syms f ss*) = *ss*
by (*simp add*: *list.map_ident_strong subset_iff*)
with *inv* **have** *p* ∈ *P* ⟹ *rename_prod g* (*rename_prod f p*) = *p* **for** *p*::('*a*,'*t*) *prod*
by(*cases p*)(*auto simp*: *Nts_def Syms_def*)
then have *inv_rename_prods*: *rename_Prods g* (*rename_Prods f P*) = *P*
using *image_iff* **by** *fastforce*
then show *?l* ⟹ *?r*
using *rename_preserves_deriven inv_rename_syms* **by** (*metis Un_upper2 le_supI1*)
qed

lemma *rename_derives_iff_inj*:
assumes *inj_on f* (*Nts P* ∪ *nts_syms α* ∪ *nts_syms β*)
shows *rename_Prods f P* ⊢ *rename_syms f α* ⇒∗ *rename_syms f β* ⟷ *P* ⊢ *α* ⇒∗ *β*
by (*meson assms relpowp_imp_rtranclp rename_deriven_iff_inj rtranclp_imp_relpowp*)

lemma *rename_deriveln_iff_inj*:
fixes *P* :: ('*a*,'*t*)*Prods*
assumes *inj_on f* (*Nts P* ∪ *nts_syms α* ∪ *nts_syms β*)
shows *rename_Prods f P* ⊢ *rename_syms f α* ⇒*l*(*n*) *rename_syms f β* ⟷ *P* ⊢ *α* ⇒*l*(*n*) *β* (**is** *?l* ⟷ *?r*)
proof
show *?r* ⟹ *?l* **by** (*rule rename_preserves_deriveln*)
next
let *?M* = *Nts P* ∪ *nts_syms α* ∪ *nts_syms β*
obtain *g* **where** *g* = *the_inv_into ?M f* **and** *inv*: (⋀*x*. *x* ∈ *?M* ⟹ (*g* (*f x*) = *x*))
using *assms* **by** (*simp add*: *the_inv_into_f_f inj_on_Un*)
then have *s* ∈ *Syms P* ∪ *set α* ∪ *set β* ⟹ *rename_sym g* (*rename_sym f s*) = *s* **for** *s*::('*a*,'*t*) *sym*
by (*cases s*) (*auto simp*: *Nts_def Syms_def nts_syms_def*)
then have *inv_rename_syms*: ⋀(*ss*::('*a*,'*t*) *syms*). *set ss* ⊆ *Syms P* ∪ *set α* ∪

set $\beta \Longrightarrow$ *rename_syms g* (*rename_syms f ss*) = *ss*
by (*simp add*: *list.map_ident_strong subset_iff*)
with *inv* **have** $p \in P \Longrightarrow$ *rename_prod g* (*rename_prod f p*) = *p* **for** *p*::('*a*,'*t*) *prod*
by(*cases p*)(*auto simp*: *Nts_def Syms_def*)
then have *inv_rename_prods*: *rename_Prods g* (*rename_Prods f P*) = *P*
using *image_iff* **by** *fastforce*
then show *?l* $\Longrightarrow$ *?r*
using *rename_preserves_deriveln inv_rename_syms* **by** (*metis Un_upper2 le_supI1*)
qed

lemma *rename_derivels_iff_inj*:
assumes *inj_on f* (*Nts P* ∪ *nts_syms* α ∪ *nts_syms* β)
shows *rename_Prods f P* ⊢ *rename_syms f* α ⇒*l*∗ *rename_syms f* β ⟷ *P* ⊢ α ⇒*l*∗ β
by (*meson assms relpowp_imp_rtranclp rename_deriveln_iff_inj rtranclp_imp_relpowp*)

lemma *Lang_rename_Prods*:
assumes *inj_on f* (*Nts P* ∪ {*S*})
shows *Lang* (*rename_Prods f P*) (*f S*) = *Lang P S*
proof −
from *assms rename_derives_iff_inj*[*of f P* [*Nt S*] *map Tm* _]
show *?thesis* **unfolding** *Lang_def* **by** (*simp*)
qed

lemma *derives_preserves_renaming*:
assumes *rename_Prods f P* ⊢ *rename_syms f u* ⇒∗ *fv*
shows ∃ *v*. *fv* = *rename_syms f v*
using *assms*
proof(*induction rule*: *derives_induct*)
case *base*
then show *?case* **by** *auto*
next
case (*step u A v w*)
then obtain *A'* **where** *A'_src*: *rename_syms f* [*Nt A'*] = [*Nt A*] **by** *auto*
from *step* **obtain** *drvW* **where** *rename_syms f drvW* = *u* @ [*Nt A*] @ *v* **by** *auto*
then have *uAv_split*: *u* @ *rename_syms f* [*Nt A'*] @ *v* = *rename_syms f drvW* **using** *A'_src* **by** *simp*
from *uAv_split* **obtain** *u'* **where** *u'_src*: *rename_syms f u'* = *u* **by** (*metis map_eq_append_conv*)
from *uAv_split* **obtain** *v'* **where** *v'_src*: *rename_syms f v'* = *v* **by** (*metis map_eq_append_conv*)
from *step* **obtain** *w'* **where** *rename_syms f w'* = *w* **by** *auto*
then have *u* @ *w* @ *v* = *rename_syms f* (*u'* @ *w'* @ *v'*) **using** *u'_src v'_src* **by** *auto*
then show *?case* **by** *fast*
qed

end

4 Disjoint Union of Sets of Productions

theory *Disjoint_Union_CFG*
imports
Regular−Sets.Regular_Set
Context_Free_Grammar
begin

This theory provides lemmas relevant when combining the productions of two grammars with disjoint sets of nonterminals. In particular that the languages of the nonterminals of one grammar is unchanged by adding productions involving only disjoint nonterminals.

lemma *derivel_disj_Un_if*:
assumes *Rhs_Nts P* ∩ *Lhss P′* = {}
and *P* ∪ *P′* ⊢ *u* ⇒*l v*
and *nts_syms u* ∩ *Lhss P′* = {}
shows *P* ⊢ *u* ⇒*l v* ∧ *nts_syms v* ∩ *Lhss P′* = {}
proof −
from *assms*(*2*) **obtain** *A w u′ v′* **where**
A_w: (*A*, *w*)∈(*P* ∪ *P′*)
and *u*: *u* = *map Tm u′* @ *Nt A* # *v′*
and *v*: *v* = *map Tm u′* @ *w* @ *v′*
unfolding *derivel_iff* **by** *fast*
then have (*A*,*w*) ∉ *P′* **using** *assms*(*3*) **unfolding** *nts_syms_def Lhss_def* **by** *auto*
then have (*A*,*w*) ∈ *P* **using** *A_w* **by** *blast*
with *u v* **have** (*A*, *w*) ∈ *P*
and *u*: *u* = *map Tm u′* @ *Nt A* # *v′*
and *v*: *v* = *map Tm u′* @ *w* @ *v′* **by** *auto*
then have *P* ⊢ *u* ⇒*l v* **using** *derivel.intros* **by** *fastforce*
moreover have *nts_syms v* ∩ *Lhss P′* = {}
using *u v assms* ‹(*A*, *w*) ∈ *P*› **unfolding** *nts_syms_def Nts_def Rhs_Nts_def* **by** *auto*
ultimately show *?thesis* **by** *fast*
qed

lemma *derive_disj_Un_if*:
assumes *Rhs_Nts P* ∩ *Lhss P′* = {}
and *P* ∪ *P′* ⊢ *u* ⇒ *v*
and *nts_syms u* ∩ *Lhss P′* = {}
shows *P* ⊢ *u* ⇒ *v* ∧ *nts_syms v* ∩ *Lhss P′* = {}
proof −
from *assms*(*2*) **obtain** *A w u′ v′* **where**
A_w: (*A*, *w*) ∈ *P* ∪ *P′*
and *u*: *u* = *u′* @ *Nt A* # *v′*

```
    and v: v = u' @ w @ v'
   unfolding derive_iff by fast
  then have (A,w) ∉ P' using assms(3) unfolding nts_syms_def Lhss_def by
auto
  then have (A,w) ∈ P using A_w by blast
  with u v have (A, w) ∈ P and u: u = u' @ Nt A # v' and v: v = u' @ w @
v' by auto
  then have P ⊢ u ⇒ v using derive.intros by fastforce
  moreover have nts_syms v ∩ Lhss P' = {}
   using u v assms ‹(A, w) ∈ P› unfolding nts_syms_def Nts_def Rhs_Nts_def
by auto
  ultimately show ?thesis by blast
qed

lemma deriveln_disj_Un_if:
assumes Rhs_Nts P ∩ Lhss P' = {}
shows ⟦ P ∪ P' ⊢ u ⇒l(n) v;  nts_syms u ∩ Lhss P' = {} ⟧ ⟹
  P ⊢ u ⇒l(n) v ∧ nts_syms v ∩ Lhss P' = {}
proof (induction n arbitrary: v)
  case 0
  then show ?case by simp
next
  case (Suc n')
  then obtain v' where split: P ∪ P' ⊢ u ⇒l(n') v' ∧ P ∪ P' ⊢ v' ⇒l v
    by (meson relpowp_Suc_E)
  with Suc have P ⊢ u ⇒l(n') v' ∧ nts_syms v' ∩ Lhss P' = {}
    by fast
  with Suc show ?case using assms derivel_disj_Un_if
    by (metis split relpowp_Suc_I)
qed

lemma deriven_disj_Un_if:
assumes Rhs_Nts P ∩ Lhss P' = {}
shows ⟦ P ∪ P' ⊢ u ⇒(n) v;  nts_syms u ∩ Lhss P' = {} ⟧ ⟹
  P ⊢ u ⇒(n) v ∧ nts_syms v ∩ Lhss P' = {}
proof (induction n arbitrary: v)
  case 0
  then show ?case by simp
next
  case (Suc n')
  then obtain v' where split: P ∪ P' ⊢ u ⇒(n') v' ∧ P ∪ P' ⊢ v' ⇒ v
    by (meson relpowp_Suc_E)
  with Suc have P ⊢ u ⇒(n') v' ∧ nts_syms v' ∩ Lhss P' = {}
    by fast
  with Suc show ?case using assms derive_disj_Un_if
    by (metis split relpowp_Suc_I)
qed

lemma derivel_disj_Un_iff:
```

assumes *Rhs_Nts P* ∩ *Lhss P′* = {}
and *nts_syms u* ∩ *Lhss P′* = {}
shows *P* ∪ *P′* ⊢ *u* ⇒*l v* ⟷ *P* ⊢ *u* ⇒*l v*
using *assms Un_derivel derivel_disj_Un_if* **by** *fastforce*

lemma *derive_disj_Un_iff*:
assumes *Rhs_Nts P* ∩ *Lhss P′* = {}
and *nts_syms u* ∩ *Lhss P′* = {}
shows *P* ∪ *P′* ⊢ *u* ⇒ *v* ⟷ *P* ⊢ *u* ⇒ *v*
using *assms Un_derive derive_disj_Un_if* **by** *fastforce*

lemma *deriveln_disj_Un_iff*:
assumes *Rhs_Nts P* ∩ *Lhss P′* = {}
and *nts_syms u* ∩ *Lhss P′* = {}
shows *P* ∪ *P′* ⊢ *u* ⇒*l*(*n*) *v* ⟷ *P* ⊢ *u* ⇒*l*(*n*) *v*
by (*metis Un_derivel assms*(*1*,*2*) *deriveln_disj_Un_if relpowp_mono*)

lemma *deriven_disj_Un_iff*:
assumes *Rhs_Nts P* ∩ *Lhss P′* = {}
and *nts_syms u* ∩ *Lhss P′* = {}
shows *P* ∪ *P′* ⊢ *u* ⇒(*n*) *v* ⟷ *P* ⊢ *u* ⇒(*n*) *v*
by (*metis Un_derive assms*(*1*,*2*) *deriven_disj_Un_if relpowp_mono*)

lemma *derives_disj_Un_iff*:
assumes *Rhs_Nts P* ∩ *Lhss P′* = {}
and *nts_syms u* ∩ *Lhss P′* = {}
shows *P* ∪ *P′* ⊢ *u* ⇒∗ *v* ⟷ *P* ⊢ *u* ⇒∗ *v*
by (*simp add*: *deriven_disj_Un_iff*[*OF assms*] *rtranclp_power*)

lemma *Lang_disj_Un1*:
assumes *Rhs_Nts P* ∩ *Lhss P′* = {}
and *S* ∉ *Lhss P′*
shows *Lang P S* = *Lang* (*P* ∪ *P′*) *S*
proof −
from *assms*(*2*) **have** *nts_syms* [*Nt S*] ∩ *Lhss P′* = {} **unfolding** *nts_syms_def Lhss_def* **by** *simp*
then show *?thesis* **unfolding** *Lang_def*
by (*simp add*: *derives_disj_Un_iff*[*OF assms*(*1*)])
qed

4.1 Disjoint Concatenation

lemma *Lang_concat_disj*:
assumes *Nts P1* ∩ *Nts P2* = {} *S* ∉ *Nts P1* ∪ *Nts P2* ∪ {*S1*,*S2*} *S1* ∉ *Nts P2* *S2* ∉ *Nts P1*
shows *Lang* ({(*S*, [*Nt S1*,*Nt S2*])} ∪ (*P1* ∪ *P2*)) *S* = *Lang P1 S1* @@ *Lang P2 S2*
proof −
let *?P* = {(*S*, [*Nt S1*,*Nt S2*])} ∪ (*P1* ∪ *P2*)

let *?L1 = Lang P1 S1*
let *?L2 = Lang P2 S2*
have *Lang ?P S ⊆ ?L1 @@ ?L2*
proof
fix *w*
assume *w ∈ Lang ?P S*
then have *?P ⊢ [Nt S] ⇒∗ map Tm w* **using** *Lang_def* **by** *fastforce*
then obtain *α* **where** *?P ⊢ α ⇒∗ map Tm w ∧ (S, α) ∈ ?P* **using** *derives_start1* **by** *fast*
then have *dervs*: *?P ⊢ [Nt S1, Nt S2] ⇒∗ map Tm w* **using** *assms* **unfolding** *Nts_def* **by** *auto*
then obtain *w1 w2* **where** *?P ⊢ [Nt S1] ⇒∗ map Tm w1 ?P ⊢ [Nt S2] ⇒∗ map Tm w2 w = w1 @ w2*
using *derives_append_decomp*[*of ?P* [*Nt S1*] [*Nt S2*]] **by** (*auto simp*: *map_eq_append_conv*)
then have *P1 ∪ ({(S, [Nt S1,Nt S2])} ∪ P2) ⊢ [Nt S1] ⇒∗ map Tm w1*
and *P2 ∪ ({(S, [Nt S1,Nt S2])} ∪ P1) ⊢ [Nt S2] ⇒∗ map Tm w2* **by** (*simp_all add*: *Un_commute*)
from *derives_disj_Un_iff*[*THEN iffD1*, *OF _ _ this*(*1*)]
derives_disj_Un_iff[*THEN iffD1*, *OF _ _ this*(*2*)] *assms*
have *P1 ⊢ [Nt S1] ⇒∗ map Tm w1 P2 ⊢ [Nt S2] ⇒∗ map Tm w2*
by (*auto simp*: *Nts_Lhss_Rhs_Nts*)
then have *w1 ∈ ?L1 w2 ∈ ?L2* **unfolding** *Lang_def* **by** *simp_all*
with ‹*w = _*› **show** *w ∈ ?L1 @@ ?L2* **by** *blast*
qed
moreover
have *?L1 @@ ?L2 ⊆ Lang ?P S*
proof
fix *w*
assume *w ∈ ?L1 @@ ?L2*
then obtain *w1 w2* **where** *w12_src*: *w1 ∈ ?L1 ∧ w2 ∈ ?L2 ∧ w = w1 @ w2* **using** *assms* **by** *blast*
have *P1 ⊆ ?P P2 ⊆ ?P* **by** *auto*
from *w12_src* **have** *1*: *P1 ⊢ [Nt S1] ⇒∗ map Tm w1* **and** *2*: *P2 ⊢ [Nt S2] ⇒∗ map Tm w2*
using *Lang_def* **by** *fast+*
have *?P ⊢ [Nt S] ⇒ [Nt S1, Nt S2]*
using *derive.intros*[*of S* [*Nt S1*, *Nt S2*] *?P* []] **by** *auto*
also have *?P ⊢ ... ⇒∗ map Tm w1 @ [Nt S2]* **using** *derives_append derives_mono*[*OF* ‹*P1 ⊆ ?P*›]
using *derives_append*[*OF derives_mono*[*OF* ‹*P1 ⊆ ?P*› *1*], *of* [*Nt S2*]] **by** *simp*
also have *?P ⊢ ... ⇒∗ map Tm w1 @ map Tm w2*
using *derives_prepend*[*OF derives_mono*[*OF* ‹*P2 ⊆ ?P*› *2*]] **by** *simp*
ultimately have *?P ⊢ [Nt S] ⇒∗ map Tm w* **using** *w12_src* **by** *simp*
then show *w ∈ Lang ?P S* **unfolding** *Lang_def* **by** *auto*
qed
ultimately show *?thesis* **by** *blast*
qed

4.2 Disjoint Union including start fork

lemma *derive_from_isolated_fork*:
⟦ *A* ∉ *Lhss P*; {(*A*,*α1*),(*A*,*α2*)} ∪ *P* ⊢ [*Nt A*] ⇒ *β* ⟧ ⟹ *β* = *α1* ∨ *β* = *α2*
unfolding *derive_singleton* **by**(*auto simp*: *Lhss_def*)

lemma *derives_fork_if_derives1*:
assumes *P* ⊢ [*Nt B1*] ⇒∗ *map Tm w*
shows {(*A*,[*Nt B1*]), (*A*,[*Nt B2*])} ∪ *P* ⊢ [*Nt A*] ⇒∗ *map Tm w* (**is** *?P* ⊢ _ ⇒∗ _)
proof −
have *?P* ⊢ [*Nt A*] ⇒ [*Nt B1*] **using** *derive_singleton* **by** *auto*
also have *?P* ⊢ [*Nt B1*] ⇒∗ *map Tm w* **using** *assms*
by (*meson derives_mono sup_ge2*)
finally show *?thesis* **.**
qed

lemma *derives_disj_if_derives_fork*:
assumes *A* ∉ *Nts P* ∪ {*B1*,*B2*}
and {(*A*,[*Nt B1*]), (*A*,[*Nt B2*])} ∪ *P* ⊢ [*Nt A*] ⇒∗ *map Tm w* (**is** *?P* ⊢ _ ⇒∗ _)
shows *P* ⊢ [*Nt B1*] ⇒∗ *map Tm w* ∨ *P* ⊢ [*Nt B2*] ⇒∗ *map Tm w*
proof −
obtain *β* **where** *steps*: *?P* ⊢ [*Nt A*] ⇒ *β* *?P* ⊢ *β* ⇒∗ *map Tm w*
using *converse_rtranclpE*[*OF assms*(*2*)] **by** *blast*
have *β* = [*Nt B1*] ∨ *β* = [*Nt B2*]
using *steps*(*1*) *derive_from_isolated_fork*[*of A P*] *assms*(*1*) **by**(*auto simp*: *Nts_Lhss_Rhs_Nts*)
then show *?thesis*
using *steps*(*2*) *derives_disj_Un_iff*[*of P* {(*A*,[*Nt B1*]), (*A*,[*Nt B2*])} *β*] *assms*
by(*auto simp*: *Nts_Lhss_Rhs_Nts*)
qed

lemma *Lang_distrib_eq_Un_Lang2*:
assumes *A* ∉ *Nts P* ∪ {*B1*,*B2*}
shows *Lang* ({(*A*,[*Nt B1*]),(*A*,[*Nt B2*])} ∪ *P*) *A* = (*Lang P B1* ∪ *Lang P B2*)
(**is** *Lang ?P* _ = _ **is** *?L1* = *?L2*)
proof
show *?L2* ⊆ *?L1* **unfolding** *Lang_def*
using *derives_fork_if_derives1*[*of P B1* _ *A B2*] *derives_fork_if_derives1*[*of P B2* _ *A B1*]
by (*auto simp add*: *insert_commute*)
next
show *?L1* ⊆ *?L2*
unfolding *Lang_def* **using** *derives_disj_if_derives_fork*[*OF assms*] **by** *auto*
qed

lemma *Lang_disj_Un2*:
assumes *Nts P1* ∩ *Nts P2* = {} *S* ∉ *Nts*(*P1* ∪ *P2*) ∪ {*S1*,*S2*} *S1* ∉ *Nts P2 S2* ∉ *Nts P1*
shows *Lang* ({(*S*,[*Nt S1*]), (*S*,[*Nt S2*])} ∪ (*P1* ∪ *P2*)) *S* = *Lang P1 S1* ∪ *Lang*

P2 S2
proof −
let *?P* = {(*S*, [*Nt S1*]), (*S*, [*Nt S2*])} ∪ (*P1* ∪ *P2*)
have *Lang ?P S* = *Lang* (*P1* ∪ *P2*) *S1* ∪ *Lang* (*P1* ∪ *P2*) *S2*
using *Lang_distrib_eq_Un_Lang2*[*OF assms*(*2*)] **by** *simp*
moreover have *Lang* (*P1* ∪ *P2*) *S1* = *Lang P1 S1* **using** *assms*(*1*,*3*)
apply(*subst Lang_disj_Un1*[*of P1 P2 S1*]) **unfolding** *Nts_Lhss_Rhs_Nts* **by** *blast*+
moreover have *Lang* (*P2* ∪ *P1*) *S2* = *Lang P2 S2* **using** *assms*(*1*,*4*)
apply(*subst Lang_disj_Un1*[*of P2 P1 S2*]) **unfolding** *Nts_Lhss_Rhs_Nts* **by** *blast*+
ultimately show *?thesis*
by (*metis sup_commute*)
qed

end

5 Context-Free Languages

theory *Context_Free_Language*
imports
Regular−Sets.Regular_Set
Renaming_CFG
Disjoint_Union_CFG
begin

5.1 Basic Definitions

This definition depends on the type of nonterminals of the grammar.

definition *CFL* :: *'n itself* ⇒ *'t list set* ⇒ *bool* **where**
CFL (*TYPE*(*'n*)) *L* = (∃ *P S*::*'n*. *L* = *Lang P S* ∧ *finite P*)

Ideally one would existentially quantify over *'n* on the right-hand side, but we cannot quantify over types in HOL. But we can prove that the type is irrelevant because we can always use another type via renaming.

For hiding the infinite type of nonterminals:

abbreviation *cfl* :: *'a lang* ⇒ *bool* **where**
cfl L ≡ *CFL* (*TYPE*(*nat*)) *L*

5.2 Closure Properties

lemma *CFL_Un_closed*:
assumes *CFL TYPE*(*'n1*) *L1 CFL TYPE*(*'n2*) *L2*
shows *CFL TYPE*((*'n1*+*'n2*)*option*) (*L1* ∪ *L2*)
proof −
from *assms* **obtain** *P1 P2* **and** *S1* :: *'n1* **and** *S2* :: *'n2*
where *L*: *L1* = *Lang P1 S1 L2* = *Lang P2 S2* **and** *fin*: *finite P1 finite P2*
unfolding *CFL_def* **by** *blast*

```
  let ?f1 = Some o (Inl:: 'n1 ⇒ 'n1 + 'n2)
  let ?f2 = Some o (Inr:: 'n2 ⇒ 'n1 + 'n2)
  let ?P1 = rename_Prods ?f1 P1
  let ?P2 = rename_Prods ?f2 P2
  let ?S1 = ?f1 S1
  let ?S2 = ?f2 S2
  let ?P = {(None, [Nt ?S1]), (None, [Nt ?S2])} ∪ (?P1 ∪ ?P2)
  have Lang ?P None = Lang ?P1 ?S1 ∪ Lang ?P2 ?S2
    by (rule Lang_disj_Un2)(auto simp: Nts_Un in_Nts_rename_Prods)
  moreover have ... = Lang P1 S1 ∪ Lang P2 S2
  proof −
    have Lang ?P1 ?S1 = L1 unfolding ‹L1 = _›
      by (meson Lang_rename_Prods comp_inj_on inj_Inl inj_Some)
    moreover have Lang ?P2 ?S2 = L2 unfolding ‹L2 = _›
      by (meson Lang_rename_Prods comp_inj_on inj_Inr inj_Some)
    ultimately show ?thesis using L by argo
  qed
  moreover have finite ?P using fin by auto
  ultimately show ?thesis
    unfolding CFL_def using L by blast
qed

lemma CFL_concat_closed:
assumes CFL TYPE('n1) L1 and CFL TYPE('n2) L2
shows CFL TYPE(('n1 + 'n2) option) (L1 @@ L2)
proof −
  obtain P1 S1 where L1_def: L1 = Lang P1 (S1::'n1) finite P1
    using assms(1) CFL_def[of L1] by auto
  obtain P2 S2 where L2_def: L2 = Lang P2 (S2::'n2) finite P2
    using assms(2) CFL_def[of L2] by auto
  let ?f1 = Some o (Inl:: 'n1 ⇒ 'n1 + 'n2)
  let ?f2 = Some o (Inr:: 'n2 ⇒ 'n1 + 'n2)
  let ?P1 = rename_Prods ?f1 P1
  let ?P2 = rename_Prods ?f2 P2
  let ?S1 = ?f1 S1
  let ?S2 = ?f2 S2
  let ?S = None :: ('n1+'n2)option
  let ?P = {(None, [Nt ?S1, Nt ?S2])} ∪ (?P1 ∪ ?P2)
  have inj ?f1 by (simp add: inj_on_def)
  then have L1r_def: L1 = Lang ?P1 ?S1
    using L1_def Lang_rename_Prods[of ?f1 P1 S1] inj_on_def by force
  have inj ?f2 by (simp add: inj_on_def)
  then have L2r_def: L2 = Lang ?P2 ?S2
    using L2_def Lang_rename_Prods[of ?f2 P2 S2] inj_on_def by force
  have Lang ?P ?S = L1 @@ L2 unfolding L1r_def L2r_def
   by(rule Lang_concat_disj) (auto simp add: disjoint_iff in_Nts_rename_Prods)
  moreover have finite ?P using ‹finite P1› ‹finite P2› by auto
  ultimately show ?thesis unfolding CFL_def by blast
qed
```

5.3 CFG as an Equation System

A CFG can be viewed as a system of equations. The least solution is denoted by *Lang_lfp*.

definition *inst_sym* :: $('n \Rightarrow 't\ lang) \Rightarrow ('n, 't)\ sym \Rightarrow 't\ lang$ **where**
inst_sym L s = (*case s of Tm a* ⇒ {[*a*]} | *Nt A* ⇒ *L A*)

definition *concats* :: *'a lang list* ⇒ *'a lang* **where**
concats Ls = *foldr* (@@) *Ls* {[]}

definition *inst_syms* :: $('n \Rightarrow 't\ lang) \Rightarrow ('n, 't)\ syms \Rightarrow 't\ lang$ **where**
inst_syms L w = *concats* (*map* (*inst_sym L*) *w*)

definition *subst_lang* :: $('n,'t)Prods \Rightarrow ('n \Rightarrow 't\ lang) \Rightarrow ('n \Rightarrow 't\ lang)$ **where**
subst_lang P L = (λ*A*. $\bigcup$ *w* ∈ *Rhss P A*. *inst_syms L w*)

definition *Lang_lfp* :: $('n, 't)\ Prods \Rightarrow 'n \Rightarrow 't\ lang$ **where**
Lang_lfp P = *lfp* (*subst_lang P*)

Now we show that this *lfp* is a Kleene fixpoint.

lemma *inst_sym_Sup_range*: *inst_sym* (*Sup*(*range F*)) = (λ*s*. *UN i*. *inst_sym* (*F i*) *s*)
by(*auto simp*: *inst_sym_def fun_eq_iff split*: *sym.splits*)

lemma *foldr_map_mono*: *F* ≤ *G* ⟹ *foldr* (@@) (*map F xs*) *Ls* ⊆ *foldr* (@@) (*map G xs*) *Ls*
by(*induction xs*)(*auto simp add*: *le_fun_def subset_eq*)

lemma *inst_sym_mono*: *F* ≤ *G* ⟹ *inst_sym F s* ⊆ *inst_sym G s*
by (*auto simp add*: *inst_sym_def le_fun_def subset_iff split*: *sym.splits*)

lemma *foldr_conc_map_inst_sym*:
assumes *omega_chain L*
shows *foldr* (@@) (*map* (λ*s*. $\bigcup$ *i*. *inst_sym* (*L i*) *s*) *xs*) *Ls* = ($\bigcup$ *i*. *foldr* (@@) (*map* (*inst_sym* (*L i*)) *xs*) *Ls*)
proof(*induction xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
show *?case* (**is** *?l* = *?r*)
proof
show *?l* ⊆ *?r*
proof
fix *w* **assume** *w* ∈ *?l*
with *Cons* **obtain** *u v i j*
where *w* = *u* @ *v u* ∈ *inst_sym* (*L i*) *a v* ∈ *foldr* (@@) (*map* (*inst_sym* (*L j*)) *xs*) *Ls* **by**(*auto*)
then show *w* ∈ *?r*

using *omega_chain_mono*[*OF assms*, *of i max i j*] *omega_chain_mono*[*OF assms*, *of j max i j*]
inst_sym_mono foldr_map_mono[*of inst_sym* (*L j*) *inst_sym* (*L* (*max i j*)) *xs Ls*] *concI*
unfolding *le_fun_def* **by**(*simp*) *blast*
qed
next
show *?r* ⊆ *?l* **using** *Cons* **by**(*fastforce*)
qed
qed

lemma *omega_cont_Lang_lfp*: *omega_cont* (*subst_lang P*)
unfolding *omega_cont_def subst_lang_def*
proof (*safe*)
fix *L* :: *nat* ⇒ *'a* ⇒ *'b lang*
assume *o*: *omega_chain L*
show (λ*A*. ⋃ (*inst_syms* (*Sup* (*range L*)) ‘ *Rhss P A*)) = (*SUP i*. (λ*A*. ⋃ (*inst_syms* (*L i*) ‘ *Rhss P A*)))
(**is** (λ*A*. *?l A*) = (λ*A*. *?r A*))
proof
fix *A* :: *'a*
have *?l A* = ⋃(⋃ *i*. (*inst_syms* (*L i*) ‘ *Rhss P A*))
by(*auto simp*: *inst_syms_def inst_sym_Sup_range concats_def foldr_conc_map_inst_sym*[*OF o*])
also have ... = *?r A*
by(*auto*)
finally show *?l A* = *?r A* **.**
qed
qed

theorem *Lang_lfp_SUP*: *Lang_lfp P* = (*SUP n*. ((*subst_lang P*)^^*n*) (λ*A*. {}))
using *Kleene_lfp*[*OF omega_cont_Lang_lfp*] **unfolding** *Lang_lfp_def bot_fun_def*
by *blast*

5.4 *Lang_lfp = Lang*

We prove that the fixpoint characterization of the language defined by a CFG is equivalent to the standard language definition via derivations. Both directions are proved separately

lemma *inst_syms_mono*: (⋀*A*. *R A* ⊆ *R' A*) ⟹ *w* ∈ *inst_syms R α* ⟹ *w* ∈ *inst_syms R' α*
unfolding *inst_syms_def concats_def*
by (*metis* (*no_types*, *lifting*) *foldr_map_mono in_mono inst_sym_mono le_fun_def*)

lemma *omega_cont_Lang_lfp_iterates*: *omega_chain* (λ*n*. ((*subst_lang P*)^^*n*) (λ*A*. {}))
using *omega_chain_iterates*[*OF mono_if_omega_cont*, *OF omega_cont_Lang_lfp*]
unfolding *bot_fun_def* **by** *blast*

lemma *in_subst_langD_inst_syms*: $w \in$ *subst_lang P L A* $\Longrightarrow \exists \alpha.\ (A,\alpha) \in P \wedge$ $w \in$ *inst_syms L* α
unfolding *subst_lang_def inst_syms_def Rhss_def* **by** (*auto split*: *prod.splits*)

lemma *foldr_conc_conc*: *foldr* (@@) *xs* {[]} @@ *A* = *foldr* (@@) *xs A*
by (*induction xs*)(*auto simp*: *conc_assoc*)

lemma *derives_if_inst_syms*:
  $w \in$ *inst_syms* ($\lambda A.\ \{w.\ P \vdash [Nt\ A] \Rightarrow* map\ Tm\ w\}$) $\alpha \Longrightarrow P \vdash \alpha \Rightarrow* map\ Tm$ w
proof (*induction* α *arbitrary*: w)
  case *Nil*
  then show *?case* **unfolding** *inst_syms_def concats_def* **by**(*auto*)
next
  case (*Cons s* α)
  show *?case*
  proof (*cases s*)
    case (*Nt A*)
    then show *?thesis* **using** *Cons*
      unfolding *inst_syms_def concats_def inst_sym_def* **by**(*fastforce simp*: *derives_Cons_decomp*)
  next
    case (*Tm a*)
    then show *?thesis* **using** *Cons*
    unfolding *inst_syms_def concats_def inst_sym_def* **by**(*auto simp*:*derives_Tm_Cons*)
  qed
qed

lemma *derives_if_in_subst_lang*: $w \in$ (*subst_lang P*)^^*n* ($\lambda A.\ \{\}$) *A* $\Longrightarrow P \vdash$ $[Nt\ A] \Rightarrow* map\ Tm\ w$
proof(*induction n arbitrary*: *w A*)
  case *0*
  then show *?case* **by** *simp*
next
  case (*Suc n*)
  let *?L* = ((*subst_lang P*)^^*n*) ($\lambda A.\ \{\}$)
  have $*$: *?L A* $\subseteq \{w.\ P \vdash [Nt\ A] \Rightarrow* map\ Tm\ w\}$ **for** *A*
    using *Suc.IH* **by** *blast*
  obtain α **where** α: $(A,\alpha) \in P$ $w \in$ *inst_syms ?L* α
    using *in_subst_langD_inst_syms*[*OF Suc.prems*[*simplified*]] **by** *blast*
  show *?case* **using** $\alpha(1)$ *derives_if_inst_syms*[*OF inst_syms_mono*[*OF* $*$, *of* _ $\lambda A.\ A$, *OF* $\alpha(2)$]]
    by (*simp add*: *derives_Cons_rule*)
qed

lemma *derives_if_Lang_lfp*: $w \in$ *Lang_lfp P A* $\Longrightarrow P \vdash [Nt\ A] \Rightarrow* map\ Tm\ w$
  unfolding *Lang_lfp_SUP* **using** *derives_if_in_subst_lang*
  by (*metis* (*mono_tags*, *lifting*) *SUP_apply UN_E*)

lemma *Lang_lfp_subset_Lang*: *Lang_lfp P A* $\subseteq$ *Lang P A*
unfolding *Lang_def* **by**(*blast intro*:*derives_if_Lang_lfp*)

The other direction:

lemma *inst_syms_decomp*:
$[\![\forall i < length\ ws.\ ws\ !\ i \in inst\_sym\ L\ (\alpha\ !\ i);\ length\ \alpha = length\ ws\ ]\!]$
$\Longrightarrow$ *concat ws* $\in$ *inst_syms L* α
proof (*induction ws arbitrary*: α)
case *Nil*
then show *?case* **unfolding** *inst_syms_def concats_def* **by** *simp*
next
case (*Cons w ws*)
then obtain $\alpha 1$ αr **where** $*$: $\alpha = \alpha 1\ \#\ \alpha r$ **by** (*metis Suc_length_conv*)
with *Cons.prems*(*2*) **have** *length* αr = *length ws* **by** *simp*
moreover from *Cons.prems* $*$ **have** $\forall i$<*length ws. ws* ! *i* $\in$ *inst_sym L* (αr ! *i*) **by** *auto*
ultimately have *concat ws* $\in$ *inst_syms L* αr **using** *Cons.IH* **by** *blast*
moreover from *Cons.prems* $*$ **have** *w* $\in$ *inst_sym L* $\alpha 1$ **by** *fastforce*
ultimately show *?case* **unfolding** *inst_syms_def concats_def* **using** $*$ **by** *force*
qed

lemma *Lang_lfp_if_derives_aux*: *P* $\vdash$ [*Nt A*] $\Rightarrow$(*n*) *map Tm w* $\Longrightarrow$ *w* $\in$ ((*subst_lang P*)^^*n*) ($\lambda A.\ \{\}$) *A*
proof(*induction n arbitrary*: *w A rule*: *less_induct*)
case (*less n*)
show *?case*
proof (*cases n*)
case *0* **then show** *?thesis* **using** *less.prems* **by** *auto*
next
case (*Suc m*)
then obtain α **where** α_*intro*: (*A*,α) $\in$ *P P* $\vdash$ α $\Rightarrow$(*m*) *map Tm w*
by (*metis deriven_start1 less.prems nat.inject*)
then obtain *ws ms* **where** $*$:
w = *concat ws* $\wedge$ *length* α = *length ws* $\wedge$ *length* α = *length ms*
$\wedge$ *sum_list ms* = *m* $\wedge$ ($\forall i < length\ ws.\ P \vdash [\alpha\ !\ i] \Rightarrow (ms\ !\ i)\ map\ Tm\ (ws\ !\ i)$)
using *derive_decomp_Tm* **by** *metis*

have $\forall i < length\ ws.\ ws\ !\ i \in inst\_sym\ (\lambda A.\ ((subst\_lang\ P)^^m)\ (\lambda A.\ \{\})\ A)\ (\alpha\ !\ i)$
proof (*rule allI* | *rule impI*)+
fix *i*
show *i* < *length ws* $\Longrightarrow$ *ws* ! *i* $\in$ *inst_sym* ((*subst_lang P* ^^ *m*) ($\lambda A.\ \{\}$)) (α ! *i*)
unfolding *inst_sym_def*
proof (*induction* α ! *i*)
case (*Nt B*)
with $*$ **have** $**$: *ms* ! *i* $\leq$ *m*
by (*metis elem_le_sum_list*)

```
      with Suc have ms ! i < n by force
      from less.IH[OF this, of B ws ! i] Nt *
        have ws ! i ∈ (subst_lang P ^^ (ms ! i)) (λA. {}) B by fastforce
      with omega_chain_mono[OF omega_cont_Lang_lfp_iterates, OF **]
          have ws ! i ∈ (subst_lang P ^^ m) (λA. {}) B by (metis le_funD
subset_iff)
      with Nt show ?case by (metis sym.simps(5))
    next
      case (Tm a)
      with * have P ⊢ map Tm [a] ⇒(ms ! i) map Tm (ws ! i) by fastforce
      then have ws ! i ∈ {[a]} using deriven_from_TmsD by fastforce
      with Tm show ?case by (metis sym.simps(6))
    qed
  qed

  from inst_syms_decomp[OF this] * have w ∈ inst_syms ((subst_lang P ^^
m) (λA. {})) α by argo
  with α_intro have w ∈ (subst_lang P) (λA. (subst_lang P ^^ m) (λA. {})
A) A
    unfolding subst_lang_def Rhss_def by force
  with Suc show ?thesis by force
 qed
qed


lemma Lang_lfp_if_derives: P ⊢ [Nt A] ⇒* map Tm w ⟹ w ∈ Lang_lfp P A
proof −
 assume P ⊢ [Nt A] ⇒* map Tm w
 then obtain n where P ⊢ [Nt A] ⇒(n) map Tm w by (meson rtranclp_power)
 from Lang_lfp_if_derives_aux[OF this] have w ∈ ((subst_lang P)^^n) (λA.
{}) A by argo
 with Lang_lfp_SUP show w ∈ Lang_lfp P A by (metis (mono_tags, lifting)
SUP_apply UNIV_I UN_iff)
qed

theorem Lang_lfp_eq_Lang: Lang_lfp P A = Lang P A
unfolding Lang_def by(blast intro: Lang_lfp_if_derives derives_if_Lang_lfp)

end
```

6 Elimination of Unit Productions

```
theory Unit_Elimination
imports Context_Free_Grammar
begin


definition unit_prods :: ('n,'t) prods ⇒ ('n,'t) Prods where
unit_prods ps = {(l,r) ∈ set ps. ∃ A. r = [Nt A]}
```

definition *unit_rtc* :: ('*n*, '*t*) *Prods* ⇒ ('*n* × '*n*) *set* **where**
unit_rtc Ps = {(*A*,*B*). *Ps* ⊢ [*Nt A*] ⇒∗ [*Nt B*] ∧ {*A*,*B*} ⊆ *Nts Ps*}

definition *unit_rm* :: ('*n*, '*t*) *prods* ⇒ ('*n*, '*t*) *Prods* **where**
unit_rm ps = (*set ps* − *unit_prods ps*)

definition *new_prods* :: ('*n*, '*t*) *prods* ⇒ ('*n*, '*t*) *Prods* **where**
new_prods ps = {(*A*,*r*). ∃ *B*. (*B*,*r*) ∈ (*unit_rm ps*) ∧ (*A*, *B*) ∈ *unit_rtc* (*unit_prods ps*)}

definition *unit_elim_rel* :: ('*n*, '*t*) *prods* ⇒ ('*n*, '*t*) *prods* ⇒ *bool* **where**
unit_elim_rel ps ps′ ≡ *set ps*′ = (*unit_rm ps* ∪ *new_prods ps*)

definition *Unit_free* :: ('*n*, '*t*) *Prods* ⇒ *bool* **where**
Unit_free P = (∄ *A B*. (*A*,[*Nt B*]) ∈ *P*)

lemma *Unit_free_if_unit_elim_rel*: *unit_elim_rel ps ps*′ ⟹ *Unit_free* (*set ps*′)

unfolding *unit_elim_rel_def unit_rm_def new_prods_def unit_prods_def Unit_free_def*
by *simp*

lemma *unit_elim_rel_Eps_free*:
assumes *Eps_free* (*set ps*) **and** *unit_elim_rel ps ps*′
shows *Eps_free* (*set ps*′)
using *assms*
unfolding *unit_elim_rel_def Eps_free_def unit_rm_def unit_prods_def new_prods_def*
by *auto*

fun *uprods* :: ('*n*,'*t*) *prods* ⇒ ('*n*,'*t*) *prods* **where**
uprods [] = [] |
uprods (*p*#*ps*) = (*if* ∃ *A*. (*snd p*) = [*Nt A*] *then p*#*uprods ps else uprods ps*)

lemma *unit_prods_uprods*: *set* (*uprods ps*) = *unit_prods ps*
unfolding *unit_prods_def* **by** (*induction ps*) *auto*

lemma *finiteunit_prods*: *finite* (*unit_prods ps*)
using *unit_prods_uprods* **by** (*metis List.finite_set*)

definition *NtsCross* :: ('*n*, '*t*) *Prods* ⇒ ('*n* × '*n*) *set* **where**
NtsCross Ps = {(*A*, *B*). *A* ∈ *Nts Ps* ∧ *B* ∈ *Nts Ps* }

lemma *finite_unit_rtc*:
assumes *finite ps*

shows *finite* (*unit_rtc ps*)
proof −
have *finite* (*Nts ps*)
unfolding *Nts_def* **using** *assms finite_nts_syms* **by** *auto*
hence *finite* (*NtsCross ps*)
unfolding *NtsCross_def* **by** *auto*
moreover have *unit_rtc ps* ⊆ *NtsCross ps*
unfolding *unit_rtc_def NtsCross_def* **by** *blast*
ultimately show *?thesis*
using *assms infinite_super* **by** *fastforce*
qed

definition *nPSlambda* :: ('*n*, '*t*) *Prods* ⇒ ('*n* × '*n*) ⇒ ('*n*, '*t*) *Prods* **where**
nPSlambda Ps d = {*fst d*} × {*r*. (*snd d*, *r*) ∈ *Ps*}

lemma *npsImage*: ⋃((*nPSlambda* (*unit_rm ps*)) ‘ (*unit_rtc* (*unit_prods ps*))) = *new_prods ps*
unfolding *new_prods_def nPSlambda_def* **by** *fastforce*

lemma *finite_nPSlambda*:
assumes *finite Ps*
shows *finite* (*nPSlambda Ps d*)
proof −
have {(*B*, *r*). (*B*, *r*) ∈ *Ps* ∧ *B* = *snd d*} ⊆ *Ps*
by *blast*
hence *finite* {(*B*, *r*). (*B*, *r*) ∈ *Ps* ∧ *B* = *snd d*}
using *assms finite_subset* **by** *blast*
hence *finite* (*snd* ‘ {(*B*, *r*). (*B*, *r*) ∈ *Ps* ∧ *B* = *snd d*})
by *simp*
moreover have {*r*. (*snd d*, *r*) ∈ *Ps*} = (*snd* ‘ {(*B*, *r*). (*B*, *r*) ∈ *Ps* ∧ *B* = *snd d*})
by *force*
ultimately show *?thesis*
using *assms* **unfolding** *nPSlambda_def* **by** *simp*
qed

lemma *finite_new_prods*: *finite* (*new_prods ps*)
proof −
have *finite* (*unit_rm ps*)
unfolding *unit_rm_def* **using** *finiteunit_prods* **by** *blast*
moreover have *finite* (*unit_rtc* (*unit_prods ps*))
using *finiteunit_prods finite_unit_rtc* **by** *blast*
ultimately show *?thesis*
using *npsImage finite_nPSlambda finite_UN* **by** *metis*
qed

lemma *finiteunit_elim_relRules*: *finite* (*unit_rm ps* ∪ *new_prods ps*)
proof −

have *finite* (*unit_rm ps*)
unfolding *unit_rm_def* **using** *finiteunit_prods* **by** *blast*
moreover have *finite* (*new_prods ps*)
using *finite_new_prods* **by** *blast*
ultimately show *?thesis* **by** *blast*
qed

lemma *unit_elim_rel_exists*: $\forall ps.\ \exists ps'.$ *unit_elim_rel ps ps'*
unfolding *unit_elim_rel_def* **using** *finite_list*[*OF finiteunit_elim_relRules*] **by** *blast*

definition *unit_elim* **where**
unit_elim ps = (*SOME ps'. unit_elim_rel ps ps'*)

lemma *unit_elim_rel_unit_elim*: *unit_elim_rel ps* (*unit_elim ps*)
by (*simp add*: *someI_ex unit_elim_def unit_elim_rel_exists*)

lemma *inNonUnitProds*:
$p \in$ *unit_rm ps* $\Longrightarrow p \in$ *set ps*
unfolding *unit_rm_def* **by** *blast*

lemma *psubDeriv*:
assumes $ps \vdash u \Rightarrow v$
and $\forall p \in ps.\ p \in ps'$
shows $ps' \vdash u \Rightarrow v$
using *assms* **by** (*meson derive_iff*)

lemma *psubRtcDeriv*:
assumes $ps \vdash u \Rightarrow* v$
and $\forall p \in ps.\ p \in ps'$
shows $ps' \vdash u \Rightarrow* v$
using *assms* **by** (*induction rule*: *rtranclp.induct*) (*auto simp*: *psubDeriv rtranclp.rtrancl_into_rtrancl*)

lemma *unit_prods_deriv*:
assumes *unit_prods ps* $\vdash u \Rightarrow* v$
shows *set ps* $\vdash u \Rightarrow* v$
proof −
have $\forall p \in$ *unit_prods ps*. $p \in$ *set ps*
unfolding *unit_prods_def* **by** *blast*
thus *?thesis*
using *assms psubRtcDeriv* **by** *blast*
qed

lemma *unit_elim_rel_r3*:
assumes *unit_elim_rel ps ps'* **and** *set ps'* $\vdash u \Rightarrow v$
shows *set ps* $\vdash u \Rightarrow* v$

```
proof −
  obtain A α r1 r2 where A: (A, α) ∈ set ps' ∧ u = r1 @ [Nt A] @ r2 ∧ v = r1
@ α @ r2
    using assms derive.cases by meson
  hence (A, α) ∈ unit_rm ps ∨ (A, α) ∈ new_prods ps
    using assms(1) unfolding unit_elim_rel_def by simp
  thus ?thesis
  proof
    assume (A, α) ∈ unit_rm ps
    hence (A, α) ∈ set ps
      using inNonUnitProds by blast
    hence set ps ⊢ r1 @ [Nt A] @ r2 ⇒ r1 @ α @ r2
      by (auto simp: derive.simps)
    thus ?thesis using A by simp
  next
    assume (A, α) ∈ new_prods ps
     from this obtain B where B: (B, α) ∈ unit_rm ps ∧ (A, B) ∈ unit_rtc
(unit_prods ps)
      unfolding new_prods_def by blast
    hence unit_prods ps ⊢ [Nt A] ⇒* [Nt B]
      unfolding unit_rtc_def by simp
    hence set ps ⊢ [Nt A] ⇒* [Nt B]
      using unit_prods_deriv by blast
    hence 1: set ps ⊢ r1 @ [Nt A] @ r2 ⇒* r1 @ [Nt B] @ r2
      using derives_append derives_prepend by blast
    have (B, α) ∈ set ps
      using B inNonUnitProds by blast
    hence set ps ⊢ r1 @ [Nt B] @ r2 ⇒ r1 @ α @ r2
      by (auto simp: derive.simps)
    thus ?thesis
      using 1 A by simp
  qed
qed

lemma unit_elim_rel_r4:
  assumes set ps' ⊢ u ⇒* v
    and unit_elim_rel ps ps'
  shows set ps ⊢ u ⇒* v
  using assms by (induction rule: rtranclp.induct) (auto simp: unit_elim_rel_r3
rtranclp_trans)

lemma deriv_unit_rtc:
  assumes set ps ⊢ [Nt A] ⇒ [Nt B]
  shows (A, B) ∈ unit_rtc (unit_prods ps)
proof −
  have (A, [Nt B]) ∈ set ps
    using assms by (simp add: derive_singleton)
  hence (A, [Nt B]) ∈ unit_prods ps
    unfolding unit_prods_def by blast
```

hence *unit_prods ps* ⊢ [*Nt A*] ⇒ [*Nt B*]
by (*simp add*: *derive_singleton*)
moreover have *B* ∈ *Nts* (*unit_prods ps*) ∧ *A* ∈ *Nts* (*unit_prods ps*)
using ‹(*A*, [*Nt B*]) ∈ *unit_prods ps*› *Nts_def nts_syms_def* **by** *fastforce*
ultimately show *?thesis*
unfolding *unit_rtc_def* **by** *blast*
qed

lemma *unit_elim_rel_r12*:
assumes *unit_elim_rel ps ps'* (*A*, *α*) ∈ *set ps'*
shows (*A*, *α*) ∉ *unit_prods ps*
using *assms* **unfolding** *unit_elim_rel_def unit_rm_def unit_prods_def new_prods_def*
by *blast*

lemma *unit_elim_rel_r14*:
assumes *unit_elim_rel ps ps'*
and *set ps* ⊢ [*Nt A*] ⇒ [*Nt B*] *set ps'* ⊢ [*Nt B*] ⇒ *v*
shows *set ps'* ⊢ [*Nt A*] ⇒ *v*
proof −
have *1*: (*A*, *B*) ∈ *unit_rtc* (*unit_prods ps*)
using *deriv_unit_rtc assms*(*2*) **by** *fast*
have *2*: (*B*, *v*) ∈ *set ps'*
using *assms*(*3*) **by** (*simp add*: *derive_singleton*)
thus *?thesis*
proof (*cases* (*B*, *v*) ∈ *set ps*)
case *True*
hence (*B*, *v*) ∈ *unit_rm ps*
unfolding *unit_rm_def* **using** *assms*(*1*) *assms*(*3*) *unit_elim_rel_r12*[*of ps ps' B v*] **by** (*simp add*: *derive_singleton*)
then show *?thesis*
using *1 assms*(*1*) **unfolding** *unit_elim_rel_def new_prods_def derive_singleton*
by *blast*
next
case *False*
hence (*B*, *v*) ∈ *new_prods ps*
using *assms*(*1*) *2* **unfolding** *unit_rm_def unit_elim_rel_def* **by** *simp*
from *this* **obtain** *C* **where** *C*: (*C*, *v*) ∈ *unit_rm ps* ∧ (*B*, *C*) ∈ *unit_rtc* (*unit_prods ps*)
unfolding *new_prods_def* **by** *blast*
hence *unit_prods ps* ⊢ [*Nt A*] ⇒∗ [*Nt C*]
using *1* **unfolding** *unit_rtc_def* **by** *auto*
hence (*A*, *C*) ∈ *unit_rtc* (*unit_prods ps*)
unfolding *unit_rtc_def* **using** *1 C unit_rtc_def* **by** *fastforce*
hence (*A*, *v*) ∈ *new_prods ps*
unfolding *new_prods_def* **using** *C* **by** *blast*
hence (*A*, *v*) ∈ *set ps'*
using *assms*(*1*) **unfolding** *unit_elim_rel_def* **by** *blast*
thus *?thesis* **by** (*simp add*: *derive_singleton*)
qed

qed

lemma *unit_elim_rel_r20_aux*:
assumes *set ps* ⊢ *l* @ [*Nt A*] @ *r* ⇒∗ *map Tm v*
shows ∃*α*. *set ps* ⊢ *l* @ [*Nt A*] @ *r* ⇒ *l* @ *α* @ *r* ∧ *set ps* ⊢ *l* @ *α* @ *r* ⇒∗ *map Tm v* ∧ (*A*, *α*) ∈ *set ps*
proof −
obtain *l′ w r′* **where** *w*: *set ps* ⊢ *l* ⇒∗ *l′* ∧ *set ps* ⊢ [*Nt A*] ⇒∗ *w* ∧ *set ps* ⊢ *r* ⇒∗ *r′* ∧ *map Tm v* = *l′* @ *w* @ *r′*
using *assms*(*1*) **by** (*metis derives_append_decomp*)
have *Nt A* ∉ *set* (*map Tm v*)
using *assms*(*1*) **by** *auto*
hence [*Nt A*] ≠ *w*
using *w* **by** *auto*
from *this* **obtain** *α* **where** *α*: *set ps* ⊢ [*Nt A*] ⇒ *α* ∧ *set ps* ⊢ *α* ⇒∗ *w*
by (*metis w converse_rtranclpE*)
hence (*A*, *α*) ∈ *set ps*
by (*simp add*: *derive_singleton*)
thus *?thesis* **by** (*metis α w derive.intros derives_append_decomp*)
qed

lemma *unit_elim_rel_r20*:
assumes *set ps* ⊢ *u* ⇒∗ *map Tm v unit_elim_rel ps ps′*
shows *set ps′* ⊢ *u* ⇒∗ *map Tm v*
using *assms* **proof** (*induction rule*: *converse_derives_induct*)
case *base*
then show *?case* **by** *blast*
next
case (*step l A r w*)
then show *?case*
proof (*cases* (*A*, *w*) ∈ *unit_prods ps*)
case *True*
from *this* **obtain** *B* **where** *w* = [*Nt B*]
unfolding *unit_prods_def* **by** *blast*
have *set ps′* ⊢ *l* @ *w* @ *r* ⇒∗ *map Tm v* ∧ *Nt B* ∉ *set* (*map Tm v*)
using *step.IH assms*(*2*) **by** *auto*
obtain *α* **where** *α*: *set ps′* ⊢ *l* @ [*Nt B*] @ *r* ⇒ *l* @ *α* @ *r* ∧ *set ps′* ⊢ *l* @ *α* @ *r* ⇒∗ *map Tm v* ∧ (*B*, *α*) ∈ *set ps′*
using *assms*(*2*) *step.IH* ‹*w*=_› *unit_elim_rel_r20_aux*[*of ps′ l B r v*] **by** *blast*
hence (*A*, *α*) ∈ *set ps′*
using *assms*(*2*) *step.hyps*(*2*) ‹*w*=_› *unit_elim_rel_r14*[*of ps ps′ A B α*] **by** (*simp add*: *derive_singleton*)
hence *set ps′* ⊢ *l* @ [*Nt A*] @ *r* ⇒∗ *l* @ *α* @ *r*
using *derive.simps* **by** *fastforce*
then show *?thesis*
using *α* **by** *auto*
next
case *False*

hence $(A, w) \in$ *unit_rm ps*
unfolding *unit_rm_def* **using** *step.hyps*(2) **by** *blast*
hence $(A, w) \in$ *set ps'*
using *assms*(2) **unfolding** *unit_elim_rel_def* **by** *simp*
hence *set ps'* $\vdash$ l @ [*Nt A*] @ $r \Rightarrow$ l @ w @ r
by (*auto simp*: *derive.simps*)
then show *?thesis*
using *step* **by** *simp*
qed
qed

theorem *unit_elim_rel_lang_eq*: *unit_elim_rel ps ps'* $\Longrightarrow$ *lang ps' S* = *lang ps S*
unfolding *Lang_def* **using** *unit_elim_rel_r4 unit_elim_rel_r20* **by** *blast*

corollary *lang_unit_elim*: *lang* (*unit_elim ps*) *A* = *lang ps A*
by (*rule unit_elim_rel_lang_eq*[*OF unit_elim_rel_unit_elim*])

end

7 Elimination of Epsilon Productions

theory *Epsilon_Elimination*
imports *Context_Free_Grammar*
begin

inductive *nullable* :: ('*n*,'*t*) *prods* $\Rightarrow$ ('*n*,'*t*) *sym* $\Rightarrow$ *bool*
for *ps* **where**
NullableSym:
$[\![(A, w) \in$ *set ps*; $\forall s \in$ *set w*. *nullable ps s*$]\!]$
$\Longrightarrow$ *nullable ps* (*Nt A*)

abbreviation *nullables ps w* $\equiv$ ($\forall s \in$ *set w*. *nullable ps s*)

lemma *nullables_if*:
assumes *set ps* $\vdash$ $u \Rightarrow* v$
and u=[a] *nullables ps v*
shows *nullables ps u*
using *assms*
proof(*induction arbitrary*: *a rule*: *rtranclp.induct*)
case (*rtrancl_refl a*)
then show *?case* **by** *simp*
next
case (*rtrancl_into_rtrancl u v w*)
from ‹*set ps* $\vdash$ $v \Rightarrow w$› **obtain** A α l r **where** $A\alpha$: $v = l$ @ [*Nt A*] @ $r \wedge w = l$ @ α @ $r \wedge (A, \alpha) \in$ *set ps*
by (*auto simp*: *derive.simps*)
from *this* ‹*nullables ps w*› **have** *nullables ps* $\alpha \wedge$ *nullables ps l* $\wedge$ *nullables ps r*
by *simp*

hence *nullables ps* [*Nt A*]
using *Aα nullable.simps* **by** *auto*
from *this* ‹*nullables ps α* ∧ *nullables ps l* ∧ *nullables ps r*› **have** *nullables ps v*
using *Aα* **by** *auto*
thus *?case*
using *rtrancl_into_rtrancl.IH rtrancl_into_rtrancl.prems*(*1*) **by** *blast*
qed

lemma *nullable_if*: *set ps* ⊢ [*a*] ⇒∗ [] ⟹ *nullable ps a*
using *nullables_if*[*of ps* [*a*] [] *a*] **by** *simp*

lemma *nullable_aux*: ∀ *s*∈*set gamma*. *nullable ps s* ∧ *set ps* ⊢ [*s*] ⇒∗ [] ⟹ *set ps* ⊢ *gamma* ⇒∗ []
proof (*induction gamma*)
case (*Cons a list*)
hence *set ps* ⊢ *list* ⇒∗ []
by *simp*
moreover have *set ps* ⊢ [*a*] ⇒∗ []
using *Cons* **by** *simp*
ultimately show *?case*
using *derives_Cons*[*of* ‹*set ps*› *list* ‹[]› ‹*a*›] **by** *simp*
qed *simp*

lemma *if_nullable*: *nullable ps a* ⟹ *set ps* ⊢ [*a*] ⇒∗ []
proof (*induction rule*: *nullable.induct*)
case (*NullableSym x gamma*)
hence *set ps* ⊢ [*Nt x*] ⇒∗ *gamma*
using *derive_singleton* **by** *blast*
also have *set ps* ⊢ *gamma* ⇒∗ []
using *NullableSym nullable_aux* **by** *blast*
finally show *?case* **.**
qed

corollary *nullable_iff*: *nullable ps a* ⟷ *set ps* ⊢ [*a*] ⇒∗ []
by (*auto simp*: *nullable_if if_nullable*)

fun *eps_closure* :: ('*n*, '*t*) *prods* ⇒ ('*n*, '*t*) *syms* ⇒ ('*n*, '*t*) *syms list* **where**
eps_closure ps [] = [[]] |
eps_closure ps (*s*#*sl*) = (
if nullable ps s then (*map* ((#) *s*) (*eps_closure ps sl*)) @ *eps_closure ps sl*
else map ((#) *s*) (*eps_closure ps sl*))

definition *eps_elim* :: ('*n*, '*t*) *prods* ⇒ ('*n*, '*t*) *Prods* **where**
eps_elim ps ≡ {(*l*,*r*'). ∃ *r*. (*l*,*r*) ∈ *set ps* ∧ *r*' ∈ *set* (*eps_closure ps r*) ∧ (*r*' ≠ [])}

definition *eps_elim_rel* :: ('*n*,'*t*) *prods* ⇒ ('*n*,'*t*) *prods* ⇒ *bool* **where**
eps_elim_rel ps ps'≡ *set ps*'= *eps_elim ps*

lemma *Eps_free_if_eps_elim_rel*: *eps_elim_rel ps ps*' ⟹ *Eps_free* (*set ps*')

unfolding *eps_elim_rel_def eps_elim_def Eps_free_def* **by** *blast*

definition *eps_elim_fun* :: ('*n*, '*t*) *prods* ⇒ ('*n*, '*t*) *prod* ⇒ ('*n*, '*t*) *Prods* **where**
eps_elim_fun ps p = {(*l'*,*r'*). *l'* = *fst p* ∧ *r'* ∈ *set* (*eps_closure ps* (*snd p*)) ∧ (*r'* ≠ [])}

lemma *eps_elim_fun_eq*: *eps_elim ps* = ⋃((*eps_elim_fun ps*) ‘ *set ps*)
proof
show *eps_elim ps* ⊆ (⋃ (*eps_elim_fun ps* ‘ *set ps*))
unfolding *eps_elim_def eps_elim_fun_def* **by** *auto*
next
show ⋃((*eps_elim_fun ps*) ‘ *set ps*) ⊆ *eps_elim ps*
proof
fix *x*
assume *x* ∈ ⋃((*eps_elim_fun ps*) ‘ *set ps*)
obtain *l r'* **where** *x* = (*l*,*r'*) **by** *fastforce*
hence (*l*,*r'*) ∈ ⋃((*eps_elim_fun ps*) ‘ *set ps*)
using ‹*x* ∈ ⋃((*eps_elim_fun ps*) ‘ *set ps*)› **by** *simp*
hence *1*: ∃ *r*. *r'* ∈ *set* (*eps_closure ps r*) ∧ (*r'* ≠ []) ∧ (*l*,*r*) ∈ *set ps*
using *eps_elim_fun_def* **by** *fastforce*
from *this* **obtain** *r* **where** *r'* ∈ *set* (*eps_closure ps r*) ∧ (*l*,*r*) ∈ *set ps*
by *blast*
thus *x* ∈ *eps_elim ps* **unfolding** *eps_elim_fun_def eps_elim_def*
using *1* ‹*x* = (*l*, *r'*)› **by** *blast*
qed
qed

lemma *finite_eps_elim*: *finite* (*eps_elim ps*)
proof −
have ∀ *p* ∈ *set ps*. *finite* (*eps_elim_fun ps p*)
unfolding *eps_elim_fun_def* **by** *auto*
hence *finite* (⋃((*eps_elim_fun ps*) ‘ *set ps*))
using *finite_UN* **by** *simp*
thus *?thesis* **using** *eps_elim_fun_eq* **by** *metis*
qed

lemma *eps_elim_rel_exists*: ∀ *ps*. ∃ *ps'*. *eps_elim_rel ps ps'*
unfolding *eps_elim_rel_def* **by** (*simp add*: *finite_list finite_eps_elim*)

lemma *eps_closure_nullable*: [] ∈ *set* (*eps_closure ps w*) ⟹ *nullables ps w*
proof (*induction w*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a r*)
hence *nullable ps a*
using *image_iff*[*of* ‹[]› ‹*eps_closure ps*› ‹{*a*#*r*}›] **by** *auto*
then show *?case*

using *Cons Un_iff* **by** *auto*
qed

lemma *eps_elim_rel_1*: $r' \in$ *set* (*eps_closure ps r*) $\Longrightarrow$ *set ps* $\vdash$ *r* $\Rightarrow*$ *r'*
proof (*induction r arbitrary*: *r'*)
case (*Cons a r*)
then show *?case*
proof (*cases nullable ps a*)
case *True*
obtain *e* **where** *e*: $e \in$ *set* (*eps_closure ps r*) $\wedge$ ($r' = (a\#e) \vee r' = e$)
using *Cons.prems True* **by** *auto*
hence *1*: *set ps* $\vdash$ *r* $\Rightarrow*$ *e*
using *Cons.IH* **by** *blast*
hence *2*: *set ps* $\vdash$ [*a*]@*r* $\Rightarrow*$ [*a*]@*e*
using *e derives_prepend* **by** *blast*
have *set ps* $\vdash$ [*a*] $\Rightarrow*$ []
using *True if_nullable* **by** *blast*
hence *set ps* $\vdash$ [*a*]@*r* $\Rightarrow*$ *r*
using *derives_append* **by** *fastforce*
thus *?thesis*
using *1 2 e* **by** *force*
next
case *False*
obtain *e* **where** *e*: $e \in$ *set* (*eps_closure ps r*) $\wedge$ ($r' = (a\#e)$)
using *Cons.prems False* **by** *auto*
hence *set ps* $\vdash$ *r* $\Rightarrow*$ *e*
using *Cons.IH* **by** *simp*
hence *set ps* $\vdash$ [*a*]@*r* $\Rightarrow*$ [*a*]@*e*
using *derives_prepend* **by** *blast*
thus *?thesis*
using *e* **by** *simp*
qed
qed *simp*

lemma *eps_elim_rel_r2*:
assumes *set ps'* $\vdash$ *u* $\Rightarrow$ *v* **and** *eps_elim_rel ps ps'*
shows *set ps* $\vdash$ *u* $\Rightarrow*$ *v*
using *assms*
proof −
obtain *A* α *x y* **where** *A*: (*A*, α) $\in$ *set ps'*$\wedge$ *u* = *x* @ [*Nt A*] @ *y* $\wedge$ *v* = *x* @ α @ *y*
using *assms derive.cases* **by** *meson*
hence *1*: (*A*, α) $\in$ {(*l*,*r'*). $\exists$ *r*. (*l*,*r*) $\in$ *set ps* $\wedge$ *r'* $\in$ *set* (*eps_closure ps r*) $\wedge$ (*r'* $\neq$ [])}
using *assms*(*2*) **unfolding** *eps_elim_rel_def eps_elim_def* **by** *simp*
obtain *r* **where** *r*: (*A*, *r*) $\in$ *set ps* $\wedge$ α $\in$ *set* (*eps_closure ps r*)
using *1* **by** *blast*
hence *set ps* $\vdash$ *r* $\Rightarrow*$ α
using *eps_elim_rel_1* **by** *blast*

hence *2*: *set ps* ⊢ *x* @ *r* @ *y* ⇒∗ *x* @ α @ *y*
using *r derives_prepend derives_append* **by** *blast*
hence *set ps* ⊢ *x* @ [*Nt A*] @ *y* ⇒ *x* @ *r* @ *y*
using *r derive.simps* **by** *fast*
thus *?thesis*
using *2* **by** (*simp add*: *A*)
qed

lemma *eps_elim_rel_r3*:
assumes *set ps′* ⊢ *u* ⇒∗ *v* **and** *eps_elim_rel ps ps′*
shows *set ps* ⊢ *u* ⇒∗ *v*
using *assms* **by** (*induction v rule*: *rtranclp_induct*) (*auto simp*: *eps_elim_rel_r2 rtranclp_trans*)

lemma *eps_elim_rel_r5*: *r* ∈ *set* (*eps_closure ps r*)
by (*induction r*) *auto*

lemma *eps_elim_rel_r4*:
assumes (*l*,*r*) ∈ *set ps*
and *eps_elim_rel ps ps′*
and (*r′* ≠ [])
and *r′* ∈ *set* (*eps_closure ps r*)
shows (*l*,*r′*) ∈ *set ps′*
using *assms* **unfolding** *eps_elim_rel_def eps_elim_def* **by** *blast*

lemma *eps_elim_rel_r7*:
assumes *eps_elim_rel ps ps′*
and *set ps* ⊢ [*Nt A*] ⇒ *v*
and *v′* ∈ *set* (*eps_closure ps v*) ∧ (*v′* ≠ [])
shows *set ps′* ⊢ [*Nt A*] ⇒ *v′*
proof −
have (*A*,*v*) ∈ *set ps*
using *assms*(*2*) **by** (*simp add*: *derive_singleton*)
hence (*A*,*v′*) ∈ *set ps′*
using *assms eps_elim_rel_r4 conjE* **by** *fastforce*
thus *?thesis*
using *derive_singleton* **by** *fast*
qed

lemma *eps_elim_rel_r12a*:
assumes *x′* ∈ *set* (*eps_closure ps x*)
and *y′* ∈ *set* (*eps_closure ps y*)
shows (*x′*@*y′*) ∈ *set* (*eps_closure ps* (*x*@*y*))
using *assms* **by** (*induction x arbitrary*: *x′ y y′ rule*: *eps_closure.induct*) *auto*

lemma *eps_elim_rel_r12b*:
assumes *x′* ∈ *set* (*eps_closure ps x*)
and *y′* ∈ *set* (*eps_closure ps y*)
and *z′* ∈ *set* (*eps_closure ps z*)

shows $(x'@y'@z') \in set\ (eps\_closure\ ps\ (x@y@z))$
using *assms*
by (*induction x arbitrary*: $x'\ y\ y'\ z\ z'$ *rule*: *eps_closure.induct*) (*auto simp*: *eps_elim_rel_r12a*)

lemma *eps_elim_rel_r14*:
assumes $r' \in set\ (eps\_closure\ ps\ (x@y))$
shows $\exists x'\ y'.\ (r'=x'@y') \wedge x' \in set\ (eps\_closure\ ps\ x) \wedge y' \in set\ (eps\_closure\ ps\ y)$
using *assms*
proof (*induction x arbitrary*: $y\ r'$ *rule*: *eps_closure.induct*)
case (*2 ps s sl*)
then show *?case*
proof −
have $\exists x'\ y'.\ s \# x = x' @ y' \wedge (x' \in (\#)\ s\ `\ set\ (eps\_closure\ ps\ sl) \vee x' \in set\ (eps\_closure\ ps\ sl)) \wedge y' \in set\ (eps\_closure\ ps\ y)$
if $\bigwedge r'\ y.\ r' \in set\ (eps\_closure\ ps\ (sl @ y)) \Longrightarrow \exists x'\ y'.\ r' = x' @ y' \wedge x' \in set\ (eps\_closure\ ps\ sl) \wedge y' \in set\ (eps\_closure\ ps\ y)$
and *nullable ps s*
and $x \in set\ (eps\_closure\ ps\ (sl @ y))$
and $r' = s \# x$
for $x :: ('a, 'b)\ sym\ list$
using *that* **by** (*metis append_Cons imageI*)
moreover have $\exists x'\ y'.\ r' = x' @ y' \wedge (x' \in (\#)\ s\ `\ set\ (eps\_closure\ ps\ sl) \vee x' \in set\ (eps\_closure\ ps\ sl)) \wedge y' \in set\ (eps\_closure\ ps\ y)$
if $\bigwedge r'\ y.\ r' \in set\ (eps\_closure\ ps\ (sl @ y)) \Longrightarrow \exists x'\ y'.\ r' = x' @ y' \wedge x' \in set\ (eps\_closure\ ps\ sl) \wedge y' \in set\ (eps\_closure\ ps\ y)$
and *nullable ps s*
and $r' \in set\ (eps\_closure\ ps\ (sl @ y))$
using *that* **by** *metis*
moreover have $\exists x'\ y'.\ s \# x = x' @ y' \wedge x' \in (\#)\ s\ `\ set\ (eps\_closure\ ps\ sl) \wedge y' \in set\ (eps\_closure\ ps\ y)$
if $\bigwedge r'\ y.\ r' \in set\ (eps\_closure\ ps\ (sl @ y)) \Longrightarrow \exists x'\ y'.\ r' = x' @ y' \wedge x' \in set\ (eps\_closure\ ps\ sl) \wedge y' \in set\ (eps\_closure\ ps\ y)$
and $\neg$ *nullable ps s*
and $x \in set\ (eps\_closure\ ps\ (sl @ y))$
and $r' = s \# x$
for $x :: ('a, 'b)\ sym\ list$
using *that* **by** (*metis append_Cons imageI*)
ultimately show *?thesis*
using *2* **by** *auto*
qed
qed *simp*

lemma *eps_elim_rel_r15*:
assumes $set\ ps \vdash [Nt\ S] \Rightarrow* u$
and *eps_elim_rel ps ps'*
and $v \in set\ (eps\_closure\ ps\ u) \wedge (v \neq [])$
shows $set\ ps' \vdash [Nt\ S] \Rightarrow* v$

```
  using assms
proof (induction u arbitrary: v rule: derives_induct)
  case base
  then show ?case
    by (cases nullable ps (Nt S)) auto
next
  case (step x A y w)
  then obtain x' w' y' where
    v: (v = (x'@w'@y')) ∧ x' ∈ set (eps_closure ps x) ∧ w' ∈ set (eps_closure ps
w) ∧ y' ∈ set (eps_closure ps y)
    using step eps_elim_rel_r14 by metis
  then show ?case
  proof (cases w' = [])
    case True
      hence v = x'@y'
        using v by simp
      have [] ∈ set (eps_closure ps w)
        using True v by simp
      hence nullables ps w
        using eps_closure_nullable by blast
      hence [] ∈ set (eps_closure ps [Nt A])
        using step(2) NullableSym by fastforce
      hence (x'@y') ∈ set (eps_closure ps (x@[Nt A]@y))
        using eps_elim_rel_r12b[of x' ps x ‹[]› ‹[Nt A]› y' y] v by simp
      then show ?thesis
        using ‹v = x' @ y'› step by blast
  next
    case False
      have (x'@[Nt A]@y') ∈ set (eps_closure ps (x@[Nt A]@y))
      using eps_elim_rel_r12b[of x' ps x ‹[Nt A]› ‹[Nt A]› y' y] eps_elim_rel_r5[of
‹[Nt A]› ps] v by blast
      hence 1: set ps' ⊢ [Nt S] ⇒* (x'@[Nt A]@y')
        using step by blast
      have set ps ⊢ [Nt A] ⇒ w
        using step(2) derive_singleton by blast
      hence set ps' ⊢ [Nt A] ⇒ w'
        using eps_elim_rel_r7[of ps ps' A w w'] False step v by blast
      hence set ps' ⊢ (x'@[Nt A]@y') ⇒ (x'@w'@y')
        using derive_append derive_prepend by blast
      thus ?thesis using 1
      by (simp add: v step.prems(2))
  qed
qed

theorem eps_elim_rel_eq_if_noe:
  assumes eps_elim_rel ps ps'
    and [] ∉ lang ps S
  shows lang ps S = lang ps' S
proof
```

show *lang ps S* ⊆ *lang ps′ S*
proof
fix *x*
assume *x* ∈ *lang ps S*
have ∀ *x*. *set ps* ⊢ [*Nt S*] ⇒∗ *x* ⟶ *x* ≠ []
using *assms Lang_def* **by** *fastforce*
hence (*map Tm x*) ∈ *set* (*eps_closure ps* (*map Tm x*))
using *eps_elim_rel_r5* **by** *auto*
hence *set ps′* ⊢ [*Nt S*] ⇒∗ (*map Tm x*)
using *assms* ‹*x* ∈ *lang ps S*› *Lang_def eps_elim_rel_r15*[*of ps S* ‹*map Tm x*›] **by** *fast*
thus *x* ∈ *lang ps′ S*
using *Lang_def* ‹*x* ∈ *lang ps S*› **by** *fast*
qed
next
show *lang ps′ S* ⊆ *lang ps S*
proof
fix *x′*
assume *x′* ∈ *lang ps′ S*
show *x′* ∈ *lang ps S*
using *assms Lang_def* ‹*x′* ∈ *lang ps′ S*› *eps_elim_rel_r3*[*of ps′* ‹[*Nt S*]› ‹*map Tm x′*› *ps*] **by** *fast*
qed
qed

lemma *noe_lang_eps_elim_rel_aux*:
assumes *ps* ⊢ [*Nt S*] ⇒∗ *w w* = []
shows ∃ *A*. *ps* ⊢ [*Nt S*] ⇒∗ [*Nt A*] ∧ (*A*, *w*) ∈ *ps*
using *assms* **by** (*induction w rule*: *rtranclp_induct*) (*auto simp*: *derive.simps*)

lemma *noe_lang_eps_elim_rel*: *eps_elim_rel ps ps′* ⟹ [] ∉ *lang ps′ S*
proof (*rule ccontr*)
assume *eps_elim_rel ps ps′* ¬([] ∉ *lang ps′ S*)
hence *set ps′* ⊢ [*Nt S*] ⇒∗ *map Tm* []
using *Lang_def* **by** *fast*
hence *set ps′* ⊢ [*Nt S*] ⇒∗ []
by *simp*
hence ∃ *A*. *set ps′* ⊢ [*Nt S*] ⇒∗ [*Nt A*] ∧ (*A*, []) ∈ *set ps′*
using *noe_lang_eps_elim_rel_aux*[*of* ‹*set ps′*›] **by** *blast*
thus *False*
using ‹*eps_elim_rel ps ps′*› **unfolding** *eps_elim_rel_def eps_elim_def* **by** *blast*
qed

theorem *eps_elim_rel_lang_eq*: *eps_elim_rel ps ps′*⟹ *lang ps′ S* = *lang ps S* − {[]}
proof
assume *eps_elim_rel ps ps′*

show *lang ps′ S* ⊆ *lang ps S* − {[]}
proof
fix *w*
assume *w* ∈ *lang ps′ S*
hence *w* ∈ *lang ps′ S* − {[]}
using *noe_lang_eps_elim_rel*[*of ps*] ‹*eps_elim_rel ps ps′*› **by** *simp*
thus *w* ∈ *lang ps S* − {[]}
using ‹*eps_elim_rel ps ps′*› **by** (*auto simp*: *Lang_def eps_elim_rel_r3*)
qed
next
assume *eps_elim_rel ps ps′*
show *lang ps S* − {[]} ⊆ *lang ps′ S*
proof
fix *w*
assume *w* ∈ *lang ps S* − {[]}
hence *1*: (*map Tm w*) ≠ []
by *simp*
have *2*: *set ps* ⊢ [*Nt S*] ⇒∗ (*map Tm w*)
using ‹*w* ∈ *lang ps S* − {[]}› *Lang_def* **by** *fast*
have (*map Tm w*) ∈ *set* (*eps_closure ps* (*map Tm w*))
using ‹*w* ∈ *lang ps S* − {[]}› *eps_elim_rel_r5* **by** *blast*
hence *set ps′* ⊢ [*Nt S*] ⇒∗ (*map Tm w*)
using *1 2 eps_elim_rel_r15*[*of ps*] ‹*eps_elim_rel ps ps′*› **by** *simp*
thus *w* ∈ *lang ps′ S*
by (*simp add*: *Lang_def*)
qed
qed

end

8 Conversion to Chomsky Normal Form

theory *Chomsky_Normal_Form*
imports *Unit_Elimination Epsilon_Elimination*
begin

definition *CNF* :: ('*n*, '*t*) *Prods* ⇒ *bool* **where**
CNF P ≡ (∀(*A*,α) ∈ *P*. (∃ *B C*. α = [*Nt B*, *Nt C*]) ∨ (∃ *t*. α = [*Tm t*]))

lemma *Nts_correct*: *A* ∉ *Nts P* ⟹ (∄ *S* α. (*S*, α) ∈ *P* ∧ (*Nt A* ∈ {*Nt S*} ∪ *set* α))
unfolding *Nts_def nts_syms_def* **by** *auto*

definition *uniformize* :: '*n*::*infinite* ⇒ '*t* ⇒ '*n* ⇒ ('*n*,'*t*)*prods* ⇒ ('*n*,'*t*) *prods* ⇒ *bool* **where**
uniformize A t S ps ps′ ≡ (
∃ *l r p s*. (*l*,*r*) ∈ *set ps* ∧ (*r* = *p*@[*Tm t*]@*s*)

$\wedge$ $(p \neq [] \vee s \neq []) \wedge A \notin nts\ ps \cup \{S\}$
$\wedge$ $ps' = ((removeAll\ (l,r)\ ps)\ @\ [(A,[Tm\ t]),\ (l,\ p@[Nt\ A]@s)]))$

lemma *uniformize_Eps_free*:
assumes *Eps_free* (*set ps*)
and *uniformize A t S ps ps'*
shows *Eps_free* (*set ps'*)
using *assms* **unfolding** *uniformize_def Eps_free_def* **by** *force*

lemma *uniformize_Unit_free*:
assumes *Unit_free* (*set ps*)
and *uniformize A t S ps ps'*
shows *Unit_free* (*set ps'*)
proof −
have *1*: $(\nexists l\ A.\ (l,[Nt\ A]) \in (set\ ps))$
using *assms*(*1*) **unfolding** *Unit_free_def* **by** *simp*
obtain *l r p s* **where** *lrps*: $(l,r) \in set\ ps \wedge (r = p@[Tm\ t]@s) \wedge (p \neq [] \vee s \neq [])$
$\wedge\ set\ ps' = ((set\ ps - \{(l,r)\}) \cup \{(A,[Tm\ t]),\ (l,\ p@[Nt\ A]@s)\})$
using *assms*(*2*) *set_removeAll* **unfolding** *uniformize_def* **by** *force*
hence $\nexists l'\ A'.\ (l,[Nt\ A']) \in \{(A,[Tm\ t]),\ (l,\ p@[Nt\ A]@s)\}$
using *Cons_eq_append_conv* **by** *fastforce*
hence $\nexists l'\ A'.\ (l',[Nt\ A']) \in ((set\ ps - \{(l,r)\}) \cup \{(A,[Tm\ t]),\ (l,\ p@[Nt\ A]@s)\})$
using *1* **by** *simp*
moreover have $set\ ps' = ((set\ ps - \{(l,r)\}) \cup \{(A,[Tm\ t]),\ (l,\ p@[Nt\ A]@s)\})$
using *lrps* **by** *simp*
ultimately show *?thesis* **unfolding** *Unit_free_def* **by** *simp*
qed

definition *prodTms* :: $('n,'t)\ prod \Rightarrow nat$ **where**
$prodTms\ p \equiv (if\ length\ (snd\ p) \leq 1\ then\ 0\ else\ length\ (filter\ (isTm)\ (snd\ p)))$

definition *prodNts* :: $('n,'t)\ prod \Rightarrow nat$ **where**
$prodNts\ p \equiv (if\ length\ (snd\ p) \leq 2\ then\ 0\ else\ length\ (filter\ (isNt)\ (snd\ p)))$

fun *badTmsCount* :: $('n,'t)\ prods \Rightarrow nat$ **where**
$badTmsCount\ ps = sum\_list(map\ prodTms\ ps)$

lemma *badTmsCountSet*: $(\forall\ p \in set\ ps.\ prodTms\ p = 0) \longleftrightarrow badTmsCount\ ps = 0$
by *auto*

fun *badNtsCount* :: $('n,'t)\ prods \Rightarrow nat$ **where**
$badNtsCount\ ps = sum\_list(map\ prodNts\ ps)$

lemma *badNtsCountSet*: $(\forall\ p \in set\ ps.\ prodNts\ p = 0) \longleftrightarrow badNtsCount\ ps = 0$
by *auto*

```
definition uniform :: ('n, 't) Prods ⇒ bool where
  uniform P ≡ ∀(A, α) ∈ P. (∄t. Tm t ∈ set α) ∨ (∃t. α = [Tm t])

lemma uniform_badTmsCount:
  uniform (set ps) ⟷ badTmsCount ps = 0
proof
  assume assm: uniform (set ps)
  have ∀p ∈ set ps. prodTms p = 0
  proof
    fix p assume p ∈ set ps
    hence (∄t. Tm t ∈ set (snd p)) ∨ (∃t. snd p = [Tm t])
      using assm unfolding uniform_def by auto
    hence length (snd p) ≤ 1 ∨ (∄t. Tm t ∈ set (snd p))
      by auto
    hence length (snd p) ≤ 1 ∨ length (filter (isTm) (snd p)) = 0
      unfolding isTm_def by (auto simp: filter_empty_conv)
    thus prodTms p = 0
      unfolding prodTms_def by argo
   qed
   thus badTmsCount ps = 0
     using badTmsCountSet by blast
next
  assume assm: badTmsCount ps = 0
  have ∀p ∈ set ps. ((∄t. Tm t ∈ set (snd p)) ∨ (∃t. snd p = [Tm t]))
  proof
    fix p assume p ∈ set ps
    hence prodTms p = 0
      using assm badTmsCountSet by blast
    hence length (snd p) ≤ 1 ∨ length (filter (isTm) (snd p)) = 0
      unfolding prodTms_def by argo
    hence length (snd p) ≤ 1 ∨ (∄t. Tm t ∈ set (snd p))
      by (auto simp: isTm_def filter_empty_conv)
    hence length (snd p) = 0 ∨ length (snd p) = 1 ∨ (∄t. Tm t ∈ set (snd p))
      using order_neq_le_trans by blast
    thus (∄t. Tm t ∈ set (snd p)) ∨ (∃t. snd p = [Tm t])
      by (auto simp: length_Suc_conv)
  qed
  thus uniform (set ps)
    unfolding uniform_def by auto
qed

definition binary :: ('n, 't) Prods ⇒ bool where
  binary P ≡ ∀(A, α) ∈ P. length α ≤ 2

lemma binary_badNtsCount:
  assumes uniform (set ps) badNtsCount ps = 0
  shows binary (set ps)
proof −
  have ∀p ∈ set ps. length (snd p) ≤ 2
```

proof
fix p **assume** *assm*: $p \in set\ ps$
obtain $A\ \alpha$ **where** $(A, \alpha) = p$
using *prod.collapse* **by** *blast*
hence $((\nexists t.\ Tm\ t \in set\ \alpha) \vee (\exists t.\ \alpha = [Tm\ t])) \wedge (prodNts\ (A, \alpha) = 0)$
using *assms badNtsCountSet assm* **unfolding** *uniform_def* **by** *auto*
hence $((\nexists t.\ Tm\ t \in set\ \alpha) \vee (\exists t.\ \alpha = [Tm\ t])) \wedge (length\ \alpha \leq 2 \vee length\ (filter\ (isNt)\ \alpha) = 0)$
unfolding *prodNts_def* **by** *force*
hence $((\nexists t.\ Tm\ t \in set\ \alpha) \vee (length\ \alpha \leq 1)) \wedge (length\ \alpha \leq 2 \vee (\nexists N.\ Nt\ N \in set\ \alpha))$
by (*auto simp*: *filter_empty_conv*[*of isNt* α] *isNt_def*)
hence $length\ \alpha \leq 2$
by (*metis Suc_1 Suc_le_eq in_set_conv_nth le_Suc_eq nat_le_linear sym.exhaust*)
thus $length\ (snd\ p) \leq 2$
using ‹$(A, \alpha) = p$› **by** *auto*
qed
thus *?thesis*
by (*auto simp*: *binary_def*)
qed

lemma *count_bin_un*: $(binary\ (set\ ps) \wedge uniform\ (set\ ps)) \longleftrightarrow (badTmsCount\ ps = 0 \wedge badNtsCount\ ps = 0)$
proof
assume $binary\ (set\ ps) \wedge uniform\ (set\ ps)$
hence $badTmsCount\ ps = 0 \wedge (\forall (A, \alpha) \in set\ ps.\ length\ \alpha \leq 2)$
unfolding *binary_def* **using** *uniform_badTmsCount* **by** *blast*
thus $badTmsCount\ ps = 0 \wedge badNtsCount\ ps = 0$
by (*metis badNtsCountSet case_prodE prod.sel*(*2*) *prodNts_def*)
next
assume $badTmsCount\ ps = 0 \wedge badNtsCount\ ps = 0$
thus $binary\ (set\ ps) \wedge uniform\ (set\ ps)$
using *binary_badNtsCount uniform_badTmsCount* **by** *blast*
qed

definition *binarizeNt* :: *'n*::*infinite* ⇒ *'n* ⇒ *'n* ⇒ *'n* ⇒ (*'n*,*'t*)*prods* ⇒ (*'n*,*'t*)*prods* ⇒ *bool* **where**
$binarizeNt\ A\ B_1\ B_2\ S\ ps\ ps' \equiv ($
$\exists l\ r\ p\ s.\ (l,r) \in set\ ps \wedge (r = p@[Nt\ B_1,Nt\ B_2]@s)$
$\wedge (p \neq [] \vee s \neq []) \wedge (A \notin (nts\ ps \cup \{S\}))$
$\wedge\ ps' = ((removeAll\ (l,r)\ ps)\ @\ [(A, [Nt\ B_1,Nt\ B_2]), (l, p@[Nt\ A]@s)]))$

lemma *binarizeNt_Eps_free*:
assumes $Eps\_free\ (set\ ps)$
and $binarizeNt\ A\ B_1\ B_2\ S\ ps\ ps'$
shows $Eps\_free\ (set\ ps')$
using *assms* **unfolding** *binarizeNt_def Eps_free_def* **by** *force*

lemma *binarizeNt_Unit_free*:
assumes *Unit_free* (*set ps*)
and *binarizeNt A* B_1 B_2 *S ps ps'*
shows *Unit_free* (*set ps'*)
proof −
have *1*: ($\nexists$ *l A*. (*l*,[*Nt A*]) ∈ (*set ps*))
using *assms*(*1*) **unfolding** *Unit_free_def* **by** *simp*
obtain *l r p s* **where** *lrps*: (*l*,*r*) ∈ *set ps* ∧ (*r* = *p*@[*Nt* B_1,*Nt* B_2]@*s*) ∧ (*p* ≠ [] ∨ *s* ≠ [])
∧ (*set ps'* = ((*set ps* − {(*l*,*r*)}) ∪ {(*A*, [*Nt* B_1,*Nt* B_2]), (*l*, *p*@[*Nt A*]@*s*)}))
using *assms*(*2*) *set_removeAll* **unfolding** *binarizeNt_def* **by** *force*
hence $\nexists$ *l' A'*. (*l*,[*Nt A'*]) ∈ {(*A*, [*Nt* B_1,*Nt* B_2]), (*l*, *p*@[*Nt A*]@*s*)}
using *Cons_eq_append_conv* **by** *fastforce*
hence $\nexists$ *l' A'*. (*l'*,[*Nt A'*]) ∈ ((*set ps* − {(*l*,*r*)}) ∪ {(*A*, [*Nt* B_1,*Nt* B_2]), (*l*, *p*@[*Nt A*]@*s*)})
using *1* **by** *simp*
moreover have *set ps'* = ((*set ps* − {(*l*,*r*)}) ∪ {(*A*, [*Nt* B_1,*Nt* B_2]), (*l*, *p*@[*Nt A*]@*s*)})
using *lrps* **by** *simp*
ultimately show *?thesis* **unfolding** *Unit_free_def* **by** *simp*
qed

lemma *binarizeNt_aux1*:
assumes *binarizeNt A* B_1 B_2 *S ps ps'*
shows *A* ≠ B_1 ∧ *A* ≠ B_2
using *assms* **unfolding** *binarizeNt_def Nts_def nts_syms_def* **by** *fastforce*

lemma *derives_sub*:
assumes *P* ⊢ [*Nt A*] ⇒ *u* **and** *P* ⊢ *xs* ⇒ *p* @ [*Nt A*] @ *s*
shows *P* ⊢ *xs* ⇒∗ *p* @ *u* @ *s*
proof −
have *P* ⊢ *p* @ [*Nt A*] @ *s* ⇒∗ *p* @ *u* @ *s*
using *assms derive_append derive_prepend* **by** *blast*
thus *?thesis*
using *assms*(*2*) **by** *simp*
qed

lemma *cnf_r1Tm*:
assumes *uniformize A t S ps ps'*
and *set ps* ⊢ *lhs* ⇒ *rhs*
shows *set ps'* ⊢ *lhs* ⇒∗ *rhs*
proof −
obtain *p' s' B v* **where** *Bv*: *lhs* = *p'*@[*Nt B*]@*s'* ∧ *rhs* = *p'*@*v*@*s'* ∧ (*B*,*v*) ∈ *set ps*
using *derive.cases*[*OF assms*(*2*)] **by** *fastforce*
obtain *l r p s* **where** *lrps*: (*l*,*r*) ∈ *set ps* ∧ (*r* = *p*@[*Tm t*]@*s*) ∧ (*p* ≠ [] ∨ *s* ≠ []) ∧ (*A* ∉ *Nts* (*set ps*))
∧ *set ps'* = ((*set ps* − {(*l*,*r*)}) ∪ {(*A*,[*Tm t*]), (*l*, *p*@[*Nt A*]@*s*)})

using *assms(1) set_removeAll* **unfolding** *uniformize_def* **by** *fastforce*
thus *?thesis*
proof (*cases* (*B*, *v*) ∈ *set ps'*)
case *True*
then show *?thesis*
using *derive.intros*[*of B v*] *Bv* **by** *blast*
next
case *False*
hence *B* = *l* ∧ *v* = *p*@[*Tm t*]@*s*
by (*simp add*: *lrps Bv*)
have *1*: *set ps'* ⊢ [*Nt l*] ⇒ *p*@[*Nt A*]@*s*
using *lrps* **by** (*simp add*: *derive_singleton*)
have *set ps'* ⊢ [*Nt A*] ⇒ [*Tm t*]
using *lrps* **by** (*simp add*: *derive_singleton*)
hence *set ps'* ⊢ [*Nt l*] ⇒* *p*@[*Tm t*]@*s*
using *1 derives_sub*[*of* ‹*set ps'*›] **by** *blast*
then show *?thesis*
using *False* ‹*B* = *l* ∧ *v* = *p*@[*Tm t*]@*s*› *Bv derives_append derives_prepend*
by *blast*
qed
qed

lemma *cnf_r1Nt*:
assumes *binarizeNt A B*$_1$ *B*$_2$ *S ps ps'*
and *set ps* ⊢ *lhs* ⇒ *rhs*
shows *set ps'* ⊢ *lhs* ⇒* *rhs*
proof −
obtain *p' s' C v* **where** *Cv*: *lhs* = *p'*@[*Nt C*]@*s'* ∧ *rhs* = *p'*@*v*@*s'* ∧ (*C*,*v*) ∈ *set*
ps
using *derive.cases*[*OF assms(2)*] **by** *fastforce*
obtain *l r p s* **where** *lrps*: (*l*,*r*) ∈ *set ps* ∧ (*r* = *p*@[*Nt B*$_1$,*Nt B*$_2$]@*s*) ∧ (*p* ≠ []
∨ *s* ≠ []) ∧ (*A* ∉ *Nts* (*set ps*))
∧ (*set ps'* = ((*set ps* − {(*l*,*r*)}) ∪ {(*A*, [*Nt B*$_1$,*Nt B*$_2$]), (*l*, *p*@[*Nt A*]@*s*)}))
using *assms(1) set_removeAll* **unfolding** *binarizeNt_def* **by** *fastforce*
thus *?thesis*
proof (*cases* (*C*, *v*) ∈ *set ps'*)
case *True*
then show *?thesis*
using *derive.intros*[*of C v*] *Cv* **by** *blast*
next
case *False*
hence *C* = *l* ∧ *v* = *p*@[*Nt B*$_1$,*Nt B*$_2$]@*s*
by (*simp add*: *lrps Cv*)
have *1*: *set ps'* ⊢ [*Nt l*] ⇒ *p*@[*Nt A*]@*s*
using *lrps* **by** (*simp add*: *derive_singleton*)
have *set ps'* ⊢ [*Nt A*] ⇒ [*Nt B*$_1$,*Nt B*$_2$]
using *lrps* **by** (*simp add*: *derive_singleton*)
hence *set ps'* ⊢ [*Nt l*] ⇒* *p*@[*Nt B*$_1$,*Nt B*$_2$]@*s*
using *1 derives_sub*[*of* ‹*set ps'*›] **by** *blast*

thus *?thesis*
using *False* ‹$C = l \wedge v = p@[Nt\ B_1,Nt\ B_2]@s$› *Cv derives_append derives_prepend* **by** *blast*
qed
qed

lemma *slemma1_1*:
assumes *uniformize A t S ps ps'*
and $(A, \alpha) \in set\ ps'$
shows $\alpha = [Tm\ t]$
proof −
have $A \notin Nts\ (set\ ps)$
using *assms(1)* **unfolding** *uniformize_def* **by** *blast*
hence $\nexists \alpha.\ (A, \alpha) \in set\ ps$
unfolding *Nts_def* **by** *auto*
hence $\nexists \alpha.\ \alpha \neq [Tm\ t] \wedge (A, \alpha) \in set\ ps'$
using *assms(1)* **unfolding** *uniformize_def* **by** *auto*
thus *?thesis*
using *assms(2)* **by** *blast*
qed

lemma *slemma1_1Nt*:
assumes *binarizeNt A* $B_1\ B_2$ *S ps ps'*
and $(A, \alpha) \in set\ ps'$
shows $\alpha = [Nt\ B_1,Nt\ B_2]$
proof −
have $A \notin Nts\ (set\ ps)$
using *assms(1)* **unfolding** *binarizeNt_def* **by** *blast*
hence $\nexists \alpha.\ (A, \alpha) \in set\ ps$
unfolding *Nts_def* **by** *auto*
hence $\nexists \alpha.\ \alpha \neq [Nt\ B_1,Nt\ B_2] \wedge (A, \alpha) \in set\ ps'$
using *assms(1)* **unfolding** *binarizeNt_def* **by** *auto*
thus *?thesis*
using *assms(2)* **by** *blast*
qed

lemma *slemma4_1*:
assumes *Nt A* $\notin$ *set rhs*
shows $\forall \alpha.\ rhs = substsNt\ A\ \alpha\ rhs$
using *assms* **by** (*simp add*: *substs_skip*)

lemma *slemma4_3_1*:
assumes $lhs = A$
shows $\alpha = substsNt\ A\ \alpha\ [Nt\ lhs]$
using *assms* **by** *simp*

lemma *slemma4_4*:
assumes *uniformize A t S ps ps'*
and $(l,r) \in set\ ps$

shows *Nt A* ∉ *set r*
proof −
have *A* ∉ *Nts* (*set ps*)
using *assms*(*1*) **unfolding** *uniformize_def* **by** *blast*
hence ∄*S α*. (*S*, *α*) ∈ *set ps* ∧ (*Nt A* ∈ {*Nt S*} ∪ *set α*)
using *Nts_correct*[*of A* ‹*set ps*›] **by** *blast*
thus *?thesis*
using *assms*(*2*) **by** *blast*
qed

lemma *slemma4_4Nt*:
assumes *binarizeNt A B*$_1$ *B*$_2$ *S ps ps*′
and (*l*,*r*) ∈ *set ps*
shows (*Nt A*) ∉ *set r*
proof −
have *A* ∉ *Nts* (*set ps*)
using *assms*(*1*) **unfolding** *binarizeNt_def* **by** *blast*
hence ∄*S α*. (*S*, *α*) ∈ *set ps* ∧ (*Nt A* ∈ {*Nt S*} ∪ *set α*)
using *Nts_correct*[*of A* ‹*set ps*›] **by** *blast*
thus *?thesis*
using *assms*(*2*) **by** *blast*
qed

lemma *lemma1*:
assumes *uniformize A t S ps ps*′
and *set ps*′ ⊢ *lhs* ⇒ *rhs*
shows *substsNt A* [*Tm t*] *lhs* = *substsNt A* [*Tm t*] *rhs*
∨ *set ps* ⊢ *substsNt A* [*Tm t*] *lhs* ⇒ *substsNt A* [*Tm t*] *rhs*
proof −
obtain *l r p s* **where** *lrps*: (*l*,*r*) ∈ *set ps* ∧ (*r* = *p*@[*Tm t*]@*s*) ∧ (*p* ≠ [] ∨ *s* ≠ []) ∧ (*A* ∉ *Nts* (*set ps*))
∧ *set ps*′ = ((*set ps* − {(*l*,*r*)}) ∪ {(*A*,[*Tm t*]), (*l*, *p*@[*Nt A*]@*s*)})
using *assms*(*1*) *set_removeAll* **unfolding** *uniformize_def* **by** *fastforce*
obtain *p*′ *s*′ *u v* **where** *uv*: *lhs* = *p*′@[*Nt u*]@*s*′ ∧ *rhs* = *p*′@*v*@*s*′ ∧ (*u*,*v*) ∈ *set ps*′
using *derive.cases*[*OF assms*(*2*)] **by** *fastforce*
thus *?thesis*
proof (*cases u* = *A*)
case *True*
then show *?thesis*
proof (*cases v* = [*Tm t*])
case *True*
have *substsNt A* [*Tm t*] *lhs* = *substsNt A* [*Tm t*] *p*′ @ *substsNt A* [*Tm t*] ([*Nt A*]@*s*′)
using *uv* ‹*u* = *A*› **by** *simp*
hence *substsNt A* [*Tm t*] *lhs* = *substsNt A* [*Tm t*] *p*′ @ [*Tm t*] @ *substsNt A* [*Tm t*] *s*′
by *simp*

```
    then show ?thesis
      by (simp add: True uv)
  next
    case False
    then show ?thesis
      using True uv assms(1) slemma1_1 by fastforce
  qed
next
  case False
  then show ?thesis
  proof (cases (Nt A) ∈ set v)
    case True
    hence 1: v = p@[Nt A]@s ∧ Nt A ∉ set p ∧ Nt A ∉ set s
      using lrps uv assms slemma4_4 by fastforce
    hence substsNt A [Tm t] v = substsNt A [Tm t] p @ substsNt A [Tm t] ([Nt
A]@s)
      by simp
    hence substsNt A [Tm t] v = p @ [Tm t] @ s
      using 1 substs_append slemma4_1 slemma4_3_1 by metis
    hence 2: (u, substsNt A [Tm t] v) ∈ set ps using lrps
      using True uv assms(1) slemma4_4 by fastforce
    have substsNt A [Tm t] lhs = substsNt A [Tm t] p' @ substsNt A [Tm t] ([Nt
u]@s')
      using uv by simp
    hence 3: substsNt A [Tm t] lhs = substsNt A [Tm t] p' @ [Nt u] @ substsNt
A [Tm t] s'
      using ‹u ≠ A› by simp
     have substsNt A [Tm t] rhs = substsNt A [Tm t] p' @ substsNt A [Tm t]
(v@s')
      using uv by simp
    hence substsNt A [Tm t] rhs = substsNt A [Tm t] p' @ substsNt A [Tm t] v
@ substsNt A [Tm t] s'
      by simp
    then show ?thesis
      using 2 3 assms(2) uv derive.simps by fast
  next
    case False
    hence 1: (u, v) ∈ set ps
      using assms(1) uv ‹u ≠ A› lrps by (simp add: in_set_conv_decomp)
     have substsNt A [Tm t] lhs = substsNt A [Tm t] p' @ substsNt A [Tm t] ([Nt
u]@s')
       using uv by simp
     hence 2: substsNt A [Tm t] lhs = substsNt A [Tm t] p' @ [Nt u] @ substsNt
A [Tm t] s'
       using ‹u ≠ A› by simp
      have substsNt A [Tm t] rhs = substsNt A [Tm t] p' @ substsNt A [Tm t]
(v@s')
       using uv by simp
     hence substsNt A [Tm t] rhs = substsNt A [Tm t] p' @ substsNt A [Tm t] v
```

```
@ substsNt A [Tm t] s'
        by simp
      hence substsNt A [Tm t] rhs = substsNt A [Tm t] p' @ v @ substsNt A [Tm
t] s'
        using False slemma4_1 by fastforce
      thus ?thesis
        using 1 2 assms(2) uv derive.simps by fast
    qed
  qed
qed

lemma lemma1Nt:
  assumes binarizeNt A B1 B2 S ps ps'
    and set ps' ⊢ lhs ⇒ rhs
  shows (substsNt A [Nt B1,Nt B2] lhs = substsNt A [Nt B1,Nt B2] rhs)
        ∨ ((set ps) ⊢ (substsNt A [Nt B1,Nt B2] lhs) ⇒ substsNt A [Nt B1,Nt B2]
rhs)
proof −
  obtain l r p s where lrps: (l,r) ∈ set ps ∧ (r = p@[Nt B1,Nt B2]@s) ∧ (p ≠ []
∨ s ≠ []) ∧ (A ∉ Nts (set ps))
    ∧ (set ps' = ((set ps − {(l,r)}) ∪ {(A, [Nt B1,Nt B2]), (l, p@[Nt A]@s)}))
    using assms(1) set_removeAll unfolding binarizeNt_def by fastforce
  obtain p' s' u v where uv: lhs = p'@[Nt u]@s' ∧ rhs = p'@v@s' ∧ (u,v) ∈ set
ps'
    using derive.cases[OF assms(2)] by fastforce
  thus ?thesis
  proof (cases u = A)
    case True
    then show ?thesis
    proof (cases v = [Nt B1,Nt B2])
      case True
      have substsNt A [Nt B1,Nt B2] lhs = substsNt A [Nt B1,Nt B2] p' @ substsNt
A [Nt B1,Nt B2] ([Nt A]@s')
        using uv ‹u = A› by simp
      hence 1: substsNt A [Nt B1,Nt B2] lhs = substsNt A [Nt B1,Nt B2] p' @ [Nt
B1,Nt B2] @ substsNt A [Nt B1,Nt B2] s'
        by simp
      have substsNt A [Nt B1,Nt B2] rhs = substsNt A [Nt B1,Nt B2] p' @ substsNt
A [Nt B1,Nt B2] ([Nt B1,Nt B2]@s')
        using uv ‹u = A› True by simp
      hence substsNt A [Nt B1,Nt B2] rhs = substsNt A [Nt B1,Nt B2] p' @ [Nt
B1,Nt B2] @ substsNt A [Nt B1,Nt B2] s'
        using assms(1) binarizeNt_aux1[of A B1 B2 S ps ps'] by auto
      then show ?thesis
        using 1 by simp
    next
      case False
      then show ?thesis
        using True uv assms(1) slemma1_1Nt by fastforce
```

qed
next
case *False*
then show *?thesis*
proof (*cases* (*Nt A*) ∈ *set v*)
case *True*
have *Nt A* ∉ *set p* ∧ *Nt A* ∉ *set s*
using *lrps assms*(*1*) **by** (*metis UnI1 UnI2 set_append slemma4_4Nt*)
hence *1*: *v* = *p*@[*Nt A*]@*s* ∧ *Nt A* ∉ *set p* ∧ *Nt A* ∉ *set s*
using *True lrps uv assms slemma4_4Nt*[*of A B*$_1$ *B*$_2$ *S ps ps'*] *binarizeNt_aux1*[*of A B*$_1$ *B*$_2$ *S ps ps'*] **by** *auto*
hence *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *v* = *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *p* @ *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] ([*Nt A*]@*s*)
by *simp*
hence *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *v* = *p* @ [*Nt B*$_1$,*Nt B*$_2$] @ *s*
using *1 substs_append slemma4_1 slemma4_3_1* **by** *metis*
hence *2*: (*u*, *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *v*) ∈ *set ps*
using *True lrps uv assms*(*1*) *slemma4_4Nt* **by** *fastforce*
have *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *lhs* = *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *p'* @ *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] ([*Nt u*]@*s'*)
using *uv* **by** *simp*
hence *3*: *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *lhs* = *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *p'* @ [*Nt u*] @ *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *s'*
using ‹*u* ≠ *A*› **by** *simp*
have *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *rhs* = *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *p'* @ *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] (*v*@*s'*)
using *uv* **by** *simp*
hence *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *rhs* = *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *p'* @ *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *v* @ *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *s'*
by *simp*
then show *?thesis*
using *2 3 assms*(*2*) *uv derive.simps* **by** *fast*
next
case *False*
hence *1*: (*u*, *v*) ∈ *set ps*
using *assms*(*1*) *uv* ‹*u* ≠ *A*› *lrps* **by** (*simp add*: *in_set_conv_decomp*)
have *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *lhs* = *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *p'* @ *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] ([*Nt u*]@*s'*)
using *uv* **by** *simp*
hence *2*: *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *lhs* = *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *p'* @ [*Nt u*] @ *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *s'*
using ‹*u* ≠ *A*› **by** *simp*
have *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *rhs* = *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *p'* @ *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] (*v*@*s'*)
using *uv* **by** *simp*
hence *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *rhs* = *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *p'* @ *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *v* @ *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *s'*
by *simp*
hence *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *rhs* = *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *p'* @ *v* @

```
substsNt A [Nt B1,Nt B2] s'
        using False slemma4_1 by fastforce
      thus ?thesis
        using 1 2 assms(2) uv derive.simps by fast
    qed
  qed
qed

lemma lemma3:
  assumes set ps' ⊢ lhs ⇒* rhs
    and uniformize A t S ps ps'
  shows set ps ⊢ substsNt A [Tm t] lhs ⇒* substsNt A [Tm t] rhs
  using assms
proof (induction rhs rule: rtranclp_induct)
  case (step y z)
  then show ?case
    using lemma1[of A t S ps ps' y z] by auto
qed simp

lemma lemma3Nt:
  assumes set ps' ⊢ lhs ⇒* rhs
    and binarizeNt A B1 B2 S ps ps'
  shows set ps ⊢ substsNt A [Nt B1, Nt B2] lhs ⇒* substsNt A [Nt B1, Nt B2] rhs
  using assms
proof (induction rhs rule: rtranclp_induct)
  case (step y z)
  then show ?case
    using lemma1Nt[of A B1 B2 S ps ps' y z] by auto
qed simp

lemma lemma4:
  assumes uniformize A t S ps ps'
  shows lang ps' S ⊆ lang ps S
proof
  fix w
  assume w ∈ lang ps' S
  hence set ps' ⊢ [Nt S] ⇒* map Tm w
    unfolding Lang_def by simp
  hence set ps' ⊢ [Nt S] ⇒* map Tm w
    using assms unfolding uniformize_def by auto
  hence set ps ⊢ substsNt A [Tm t] [Nt S] ⇒* substsNt A [Tm t] (map Tm w)
    using assms lemma3[of ps' ‹[Nt S]› ‹map Tm w›] by blast
  moreover have substsNt A [Tm t] [Nt S] = [Nt S]
    using assms unfolding uniformize_def by auto
  moreover have substsNt A [Tm t] (map Tm w) = map Tm w
    by simp
  ultimately show w ∈ lang ps S
    by (simp add: Lang_def)
qed
```

lemma *lemma4Nt*:
assumes *binarizeNt A* B_1 B_2 *S ps ps'*
shows *lang ps' S* ⊆ *lang ps S*
proof
fix *w*
assume *w* ∈ *lang ps' S*
hence *set ps'* ⊢ [*Nt S*] ⇒∗ *map Tm w*
by (*simp add*: *Lang_def*)
hence *set ps'* ⊢ [*Nt S*] ⇒∗ *map Tm w*
using *assms* **unfolding** *binarizeNt_def* **by** *auto*
hence *set ps* ⊢ *substsNt A* [*Nt* B_1, *Nt* B_2] [*Nt S*] ⇒∗ *substsNt A* [*Nt* B_1, *Nt* B_2] (*map Tm w*)
using *assms lemma3Nt*[*of ps'* ‹[*Nt S*]› ‹*map Tm w*›] **by** *blast*
moreover have *substsNt A* [*Nt* B_1, *Nt* B_2] [*Nt S*] = [*Nt S*]
using *assms* **unfolding** *binarizeNt_def* **by** *auto*
moreover have *substsNt A* [*Nt* B_1, *Nt* B_2] (*map Tm w*) = *map Tm w* **by** *simp*
ultimately show *w* ∈ *lang ps S* **using** *Lang_def*
by (*metis* (*no_types*, *lifting*) *mem_Collect_eq*)
qed

lemma *slemma5_1*:
assumes *set ps* ⊢ *u* ⇒∗ *v*
and *uniformize A t S ps ps'*
shows *set ps'* ⊢ *u* ⇒∗ *v*
using *assms* **by** (*induction v rule*: *rtranclp_induct*) (*auto simp*: *cnf_r1Tm rtranclp_trans*)

lemma *slemma5_1Nt*:
assumes *set ps* ⊢ *u* ⇒∗ *v*
and *binarizeNt A* B_1 B_2 *S ps ps'*
shows *set ps'* ⊢ *u* ⇒∗ *v*
using *assms* **by** (*induction v rule*: *rtranclp_induct*) (*auto simp*: *cnf_r1Nt rtranclp_trans*)

lemma *lemma5*:
assumes *uniformize A t S ps ps'*
shows *lang ps S* ⊆ *lang ps' S*
proof
fix *w*
assume *w* ∈ *lang ps S*
hence *set ps* ⊢ [*Nt S*] ⇒∗ *map Tm w*
using *assms* **unfolding** *Lang_def uniformize_def* **by** *auto*
thus *w* ∈ *lang ps' S*
using *assms slemma5_1 Lang_def* **by** *fastforce*
qed

lemma *lemma5Nt*:
assumes *binarizeNt A* B_1 B_2 *S ps ps'*

shows *lang ps S* ⊆ *lang ps′ S*
proof
fix *w*
assume *w* ∈ *lang ps S*
hence *set ps* ⊢ [*Nt S*] ⇒∗ *map Tm w*
using *assms* **unfolding** *Lang_def binarizeNt_def* **by** *auto*
thus *w* ∈ *lang ps′ S*
using *assms slemma5_1Nt Lang_def* **by** *fast*
qed

lemma *cnf_lemma1*: *uniformize A t S ps ps′* ⟹ *lang ps S* = *lang ps′ S*
using *lemma4 lemma5* **by** *fast*

lemma *cnf_lemma1Nt*: *binarizeNt A B*$_1$ *B*$_2$ *S ps ps′* ⟹ *lang ps S* = *lang ps′ S*
using *lemma4Nt lemma5Nt* **by** *fast*

lemma *uniformizeRtc_Eps_free*:
assumes (λ*x y*. ∃ *A t*. *uniformize A t S x y*)^** *ps ps′*
and *Eps_free* (*set ps*)
shows *Eps_free* (*set ps′*)
using *assms* **by** (*induction rule*: *rtranclp_induct*) (*auto simp*: *uniformize_Eps_free*)

lemma *binarizeNtRtc_Eps_free*:
assumes (λ*x y*. ∃ *A t B*$_1$ *B*$_2$. *binarizeNt A B*$_1$ *B*$_2$ *S x y*)^** *ps ps′*
and *Eps_free* (*set ps*)
shows *Eps_free* (*set ps′*)
using *assms* **by** (*induction rule*: *rtranclp_induct*) (*auto simp*: *binarizeNt_Eps_free*)

lemma *uniformizeRtc_Unit_free*:
assumes (λ*x y*. ∃ *A t*. *uniformize A t S x y*)^** *ps ps′*
and *Unit_free* (*set ps*)
shows *Unit_free* (*set ps′*)
using *assms* **by** (*induction rule*: *rtranclp_induct*) (*auto simp*: *uniformize_Unit_free*)

lemma *binarizeNtRtc_Unit_free*:
assumes (λ*x y*. ∃ *A t B*$_1$ *B*$_2$. *binarizeNt A B*$_1$ *B*$_2$ *S x y*)^** *ps ps′*
and *Unit_free* (*set ps*)
shows *Unit_free* (*set ps′*)
using *assms* **by** (*induction rule*: *rtranclp_induct*) (*auto simp*: *binarizeNt_Unit_free*)

lemma *uniformize_Nts*:
assumes *uniformize A t S ps ps′ S* ∈ *Nts* (*set ps*)
shows *S* ∈ *Nts* (*set ps′*)
proof −
obtain *l r p s* **where** *lrps*: (*l*,*r*) ∈ *set ps* ∧ (*r* = *p*@[*Tm t*]@*s*) ∧ (*p* ≠ [] ∨ *s* ≠ []) ∧ (*A* ∉ *Nts* (*set ps*))
∧ *set ps′* = ((*set ps* − {(*l*,*r*)}) ∪ {(*A*,[*Tm t*]), (*l*, *p*@[*Nt A*]@*s*)})

using *assms(1) set_removeAll* **unfolding** *uniformize_def* **by** *fastforce*
thus *?thesis*
proof (*cases* $S \in Nts\ \{(l,r)\}$)
case *True*
hence $S \in Nts\ \{(A,[Tm\ t]), (l,\ p@[Nt\ A]@s)\}$
unfolding *Nts_def nts_syms_def* **using** *lrps* **by** *auto*
then show *?thesis* **using** *lrps Nts_Un* **by** (*metis UnCI*)
next
case *False*
hence $S \in Nts\ (set\ ps - \{(l,r)\})$
unfolding *Nts_def* **using** *lrps*
by (*metis UnCI UnE Un_Diff_cancel2 assms(2) Nts_Un Nts_def*)
then show *?thesis*
by (*simp add*: *lrps Nts_def*)
qed
qed

lemma *uniformizeRtc_Nts*:
assumes $(\lambda x\ y.\ \exists A\ t.\ uniformize\ A\ t\ S\ x\ y)^{**}\ ps\ ps'\ S \in Nts\ (set\ ps)$
shows $S \in Nts\ (set\ ps')$
using *assms* **by** (*induction rule*: *rtranclp_induct*) (*auto simp*: *uniformize_Nts*)

theorem *cnf_lemma2*:
assumes $(\lambda x\ y.\ \exists A\ t.\ uniformize\ A\ t\ S\ x\ y)^{**}\ ps\ ps'$
shows $lang\ ps\ S = lang\ ps'\ S$
using *assms* **by** (*induction rule*: *rtranclp_induct*) (*fastforce simp*: *cnf_lemma1*)+

theorem *cnf_lemma2Nt*:
assumes $(\lambda x\ y.\ \exists A\ t\ B_1\ B_2.\ binarizeNt\ A\ B_1\ B_2\ S\ x\ y)^{**}\ ps\ ps'$
shows $lang\ ps\ S = lang\ ps'\ S$
using *assms* **by** (*induction rule*: *rtranclp_induct*) (*fastforce simp*: *cnf_lemma1Nt*)+

theorem *cnf_lemma*:
assumes $(\lambda x\ y.\ \exists A\ t.\ uniformize\ A\ t\ S\ x\ y)^{**}\ ps\ ps'$
and $(\lambda x\ y.\ \exists A\ B_1\ B_2.\ binarizeNt\ A\ B_1\ B_2\ S\ x\ y)^{**}\ ps'\ ps''$
shows $lang\ ps\ S = lang\ ps''\ S$
using *assms cnf_lemma2 cnf_lemma2Nt uniformizeRtc_Nts* **by** *fastforce*

lemma *badTmsCount_append*: $badTmsCount\ (ps@ps') = badTmsCount\ ps + badTmsCount\ ps'$
by *auto*

lemma *badNtsCount_append*: $badNtsCount\ (ps@ps') = badNtsCount\ ps + badNtsCount\ ps'$
by *auto*

lemma *badTmsCount_removeAll*:
assumes *prodTms p > 0 p ∈ set ps*
shows *badTmsCount (removeAll p ps) < badTmsCount ps*
using *assms* **by** (*induction ps*) *fastforce+*

lemma *badNtsCount_removeAll*:
assumes *prodNts p > 0 p ∈ set ps*
shows *badNtsCount (removeAll p ps) < badNtsCount ps*
using *assms* **by** (*induction ps*) *fastforce+*

lemma *badTmsCount_removeAll2*:
assumes *prodTms p > 0 p ∈ set ps prodTms p' < prodTms p*
shows *badTmsCount (removeAll p ps) + prodTms p' < badTmsCount ps*
using *assms* **by** (*induction ps*) *fastforce+*

lemma *badNtsCount_removeAll2*:
assumes *prodNts p > 0 p ∈ set ps prodNts p' < prodNts p*
shows *badNtsCount (removeAll p ps) + prodNts p' < badNtsCount ps*
using *assms* **by** (*induction ps*) *fastforce+*

lemma *lemma6_a*:
assumes *uniformize A t S ps ps'* **shows** *badTmsCount (ps') < badTmsCount ps*
proof −
from *assms* **obtain** *l r p s* **where** *lrps*: *(l,r) ∈ set ps ∧ (r = p@[Tm t]@s) ∧ (p ≠ [] ∨ s ≠ []) ∧ (A ∉ Nts (set ps))*
∧ ps' = ((removeAll (l,r) ps) @ [(A,[Tm t]), (l, p@[Nt A]@s)])
unfolding *uniformize_def* **by** *auto*
hence *prodTms (l,p@[Tm t]@s) = length (filter (isTm) (p@[Tm t]@s))*
unfolding *prodTms_def* **by** *auto*
hence *1*: *prodTms (l,p@[Tm t]@s) = Suc (length (filter (isTm) (p@s)))*
by (*simp add*: *isTm_def*)
have *2*: *badTmsCount ps' = badTmsCount (removeAll (l,r) ps) + badTmsCount [(A,[Tm t])] + badTmsCount [(l, p@[Nt A]@s)]*
using *lrps* **by** (*auto simp*: *badTmsCount_append*)
have *3*: *badTmsCount (removeAll (l,r) ps) < badTmsCount ps*
using *1 badTmsCount_removeAll lrps gr0_conv_Suc* **by** *blast*
have *prodTms (l, p@[Nt A]@s) = (length (filter (isTm) (p@[Nt A]@s))) ∨ prodTms (l, p@[Nt A]@s) = 0*
unfolding *prodTms_def* **using** *lrps* **by** *simp*
thus *?thesis*
proof
assume *prodTms (l, p@[Nt A]@s) = (length (filter (isTm) (p@[Nt A]@s)))*
hence *badTmsCount ps' = badTmsCount (removeAll (l,r) ps) + prodTms (l, p@[Nt A]@s)*
using *2* **by** (*simp add*: *prodTms_def*)
moreover have *prodTms (l,p@[Nt A]@s) < prodTms (l,p@[Tm t]@s)*
using *1* ‹*prodTms (l, p @ [Nt A] @ s) = length (filter isTm (p @ [Nt A] @ s))*› *isTm_def* **by** *force*

```
    ultimately show badTmsCount ps' < badTmsCount ps
      using badTmsCount_removeAll2[of (l,r) ps (l,p @[Nt A]@s)] lrps 1 by auto
  next
    assume prodTms (l, p@[Nt A]@s) = 0
    hence badTmsCount ps' = badTmsCount (removeAll (l,r) ps)
      using 2 by (simp add: prodTms_def)
    thus badTmsCount ps' < badTmsCount ps
      using 3 by simp
  qed
qed

lemma lemma6_b:
  assumes binarizeNt A B1 B2 S ps ps' shows badNtsCount ps' < badNtsCount
ps
proof −
  from assms obtain l r p s where lrps: (l,r) ∈ set ps ∧ (r = p@[Nt B1,Nt B2]@s)
∧ (p ≠ [] ∨ s ≠ []) ∧ (A ∉ Nts (set ps))
    ∧ ps' = ((removeAll (l,r) ps) @ [(A, [Nt B1,Nt B2]), (l, p@[Nt A]@s)])
    unfolding binarizeNt_def by auto
   hence prodNts (l,p@[Nt B1,Nt B2]@s) = length (filter (isNt) (p@[Nt B1,Nt
B2]@s))
    unfolding prodNts_def by auto
   hence 1: prodNts (l,p@[Nt B1,Nt B2]@s) = Suc (Suc (length (filter (isNt)
(p@s))))
    by (simp add: isNt_def)
  have 2: badNtsCount ps' = badNtsCount (removeAll (l,r) ps) + badNtsCount
[(A, [Nt B1,Nt B2])] + badNtsCount [(l, (p@[Nt A]@s))]
    using lrps by (auto simp: badNtsCount_append prodNts_def)
  have 3: badNtsCount (removeAll (l,r) ps) < badNtsCount ps
    using lrps badNtsCount_removeAll 1 by force
  have prodNts (l, p@[Nt A]@s) = length (filter (isNt) (p@[Nt A]@s)) ∨ prodNts
(l, p@[Nt A]@s) = 0
    unfolding prodNts_def using lrps by simp
  thus ?thesis
  proof
    assume prodNts (l, p@[Nt A]@s) = length (filter (isNt) (p@[Nt A]@s))
     hence badNtsCount ps' = badNtsCount (removeAll (l,r) ps) + badNtsCount
[(l, (p@[Nt A]@s))]
      using 2 by (simp add: prodNts_def)
    moreover have prodNts (l, p@[Nt A]@s) < prodNts (l,p@[Nt B1,Nt B2]@s)
       using 1 ‹prodNts (l, p@[Nt A]@s) = length (filter (isNt) (p@[Nt A]@s))›
isNt_def by simp
    ultimately show ?thesis
     using badNtsCount_removeAll2[of (l,r) ps (l, (p@[Nt A]@s))] 1 lrps by auto
  next
    assume prodNts (l, p@[Nt A]@s) = 0
    hence badNtsCount ps' = badNtsCount (removeAll (l,r) ps)
      using 2 by (simp add: prodNts_def)
    thus ?thesis
```

```
    using 3 by simp
  qed
qed

lemma badTmsCount0_removeAll: badTmsCount ps = 0 ⟹ badTmsCount (removeAll
(l,r) ps) = 0
by auto

lemma slemma15_a:
  assumes binarizeNt A B1 B2 S ps ps'
    and badTmsCount ps = 0
  shows badTmsCount ps' = 0
proof −
  obtain l r p s where lrps: (l,r) ∈ set ps ∧ (r = p@[Nt B1,Nt B2]@s) ∧ (p ≠ []
∨ s ≠ []) ∧ (A ∉ Nts (set ps))
    ∧ (ps' = ((removeAll (l,r) ps) @ [(A, [Nt B1,Nt B2]), (l, p@[Nt A]@s)]))
    using assms(1) unfolding binarizeNt_def by auto
  hence badTmsCount ps' = badTmsCount (removeAll (l,r) ps) + badTmsCount
[(l, (p@[Nt A]@s))]
    by (auto simp: badTmsCount_append prodTms_def isTm_def)
  moreover have badTmsCount (removeAll (l,r) ps) = 0
    using badTmsCount0_removeAll[of ps l r] assms(2) by simp
  moreover have badTmsCount [(l, (p@[Nt A]@s))] = 0
  proof −
    have prodTms (l,p@[Nt B1,Nt B2]@s) = 0
      using lrps assms(2) badTmsCountSet by auto
    thus badTmsCount [(l, (p@[Nt A]@s))] = 0
      by (auto simp: isTm_def prodTms_def)
  qed
  ultimately show ?thesis
    by simp
qed

lemma lemma15_a:
  assumes (λx y. ∃ A B1 B2. binarizeNt A B1 B2 S x y)^** ps ps'
    and badTmsCount ps = 0
  shows badTmsCount ps' = 0
 using assms by (induction) (auto simp: slemma15_a simp del: badTmsCount.simps)

lemma noTms_prodTms0:
  assumes prodTms (l,r) = 0
  shows length r ≤ 1 ∨ (∀ a ∈ set r. isNt a)
proof −
  have length r ≤ 1 ∨ (∄ a. a ∈ set r ∧ isTm a)
    using assms unfolding prodTms_def using empty_filter_conv by fastforce
  thus ?thesis
    by (metis isNt_def isTm_def sym.exhaust)
qed
```

lemma *badTmsCountNot0*:
assumes *badTmsCount ps > 0*
shows $\exists\, l\ r\ t.\ (l,r) \in set\ ps \wedge length\ r \geq 2 \wedge Tm\ t \in set\ r$
proof −
have $\exists\, p \in set\ ps.\ prodTms\ p > 0$
using *assms badTmsCountSet not_gr0* **by** *blast*
from *this* **obtain** *l r* **where** *lr*: $(l,\ r) \in set\ ps \wedge prodTms\ (l,r) > 0$
by *auto*
hence *1*: $length\ r \geq 2$
unfolding *prodTms_def* **using** *not_le_imp_less* **by** *fastforce*
hence $prodTms\ (l,r) = length\ (filter\ (isTm)\ r)$
unfolding *prodTms_def* **by** *simp*
hence $\exists\, t.\ Tm\ t \in set\ r$
by (*metis lr empty_filter_conv isTm_def length_greater_0_conv*)
thus *?thesis* **using** *lr 1* **by** *blast*
qed

lemma *badNtsCountNot0*:
assumes *badNtsCount ps > 0*
shows $\exists\, l\ r.\ (l,\ r) \in set\ ps \wedge length\ r \geq 3$
proof −
have $\exists\, p \in set\ ps.\ prodNts\ p > 0$
using *assms badNtsCountSet not_gr0* **by** *blast*
from *this* **obtain** *l r* **where** *lr*: $(l,\ r) \in set\ ps \wedge prodNts\ (l,r) > 0$
by *auto*
hence $length\ r \geq 3$
unfolding *prodNts_def* **using** *not_le_imp_less* **by** *fastforce*
thus *?thesis* **using** *lr* **by** *auto*
qed

lemma *list_longer2*: $length\ l \geq 2 \wedge x \in set\ l \Longrightarrow (\exists\, hd\ tl\ .\ l = hd@[x]@tl \wedge (hd \neq [] \vee tl \neq []))$
using *split_list_last* **by** *fastforce*

lemma *list_longer3*: $length\ l \geq 3 \Longrightarrow (\exists\, hd\ tl\ x\ y.\ l = hd@[x]@[y]@tl \wedge (hd \neq [] \vee tl \neq []))$
by (*metis Suc_le_length_iff append.left_neutral append_Cons neq_Nil_conv numeral_3_eq_3*)

lemma *lemma8_a*:
assumes *badTmsCount ps > 0* **shows** $\exists\, ps'\ A\ t.\ uniformize\ A\ t\ S\ ps\ ps'$
proof −
obtain *A* **where** *A*: $A \notin nts\ ps \cup \{S\}$ **using** *ex_new_if_finite*[*OF infinite_UNIV*] *finite_nts*[*of ps*]
by *blast*
then obtain *l r t* **where** *lr*: $(l,r) \in set\ ps \wedge length\ r \geq 2 \wedge Tm\ t \in set\ r$
using *assms badTmsCountNot0* **by** *blast*
then obtain *p s* **where** *ps*: $r = p@[Tm\ t]@s \wedge (p \neq [] \vee s \neq [])$
unfolding *isTm_def* **using** *lr list_longer2*[*of r*] **by** *blast*

from *this* **obtain** *ps'* **where** *ps'* = *removeAll* (*l*, *r*) *ps* @ [(*A*, [*Tm t*]), (*l*, *p* @ [*Nt A*] @ *s*)]
by *auto*
hence *uniformize A t S ps ps'*
unfolding *uniformize_def* **using** *lr ps A* **by** *auto*
thus *?thesis* **by** *blast*
qed

lemma *lemma8_b*:
assumes *badTmsCount ps* = *0* **and** *badNtsCount ps* > *0*
shows ∃ *ps' A B*$_1$ *B*$_2$. *binarizeNt A B*$_1$ *B*$_2$ *S ps ps'*
proof −
obtain *l r* **where** *lr*: (*l*, *r*) ∈ *set ps* ∧ *length r* ≥ *3*
using *assms*(*2*) *badNtsCountNot0* **by** *blast*
obtain *A* **where** *A*: *A* ∉ *nts ps* ∪ {*S*} **using** *ex_new_if_finite*[*OF infinite_UNIV*] *finite_nts*[*of ps*]
by *blast*
obtain *p s X Y* **where** *psXY*: *r* = *p*@[*X*]@[*Y*]@*s* ∧ (*p* ≠ [] ∨ *s* ≠ [])
using *lr list_longer3*[*of r*] **by** *blast*
have (∀ *a* ∈ *set r*. *isNt a*)
using *lr assms*(*1*) *badTmsCountSet*[*of ps*] *noTms_prodTms0*[*of l r*] **by** *fastforce*
from *this* **obtain** *B*$_1$ *B*$_2$ **where** *X* = *Nt B*$_1$ ∧ *Y* = *Nt B*$_2$
using *isNt_def psXY* **by** *fastforce*
hence *B*: (*r* = *p*@[*Nt B*$_1$,*Nt B*$_2$]@*s*) ∧ (*p* ≠ [] ∨ *s* ≠ [])
using *psXY* **by** *auto*
hence *binarizeNt A B*$_1$ *B*$_2$ *S ps* (*removeAll* (*l*, *r*) *ps* @ [(*A*, [*Nt B*$_1$, *Nt B*$_2$]), (*l*, *p* @ [*Nt A*] @ *s*)])
unfolding *binarizeNt_def* **using** *A lr B* **by** *auto*
thus *?thesis* **by** *blast*
qed

lemma *uniformize_2*: ∃ *ps'*. (λ*x y*. ∃ *A t*. *uniformize A t S x y*)^** *ps ps'* ∧ (*badTmsCount ps'* = *0*)
proof (*induction badTmsCount ps arbitrary*: *ps rule*: *less_induct*)
case *less*
then show *?case*
proof (*cases badTmsCount ps* = *0*)
case *False*
from *this* **obtain** *ps' A t* **where** *g'*: *uniformize A t S ps ps'*
using *lemma8_a* **by** *blast*
hence *badTmsCount ps'* < *badTmsCount ps*
using *lemma6_a*[*of A t*] **by** *blast*
from *this* **obtain** *ps''* **where** (λ*x y*. ∃ *A t*. *uniformize A t S x y*)** *ps' ps''* ∧ *badTmsCount ps''* = *0*
using *less* **by** *blast*
thus *?thesis*
using *g' converse_rtranclp_into_rtranclp*[*of* (λ*x y*. ∃ *A t*. *uniformize A t S x y*) *ps ps' ps''*] **by** *blast*
qed *blast*

qed

lemma *binarizeNt_2*:
assumes *badTmsCount ps = 0*
shows $\exists ps'.\ (\lambda x\ y.\ \exists A\ B_1\ B_2.\ binarizeNt\ A\ B_1\ B_2\ S\ x\ y)^{**}\ ps\ ps' \wedge (badNtsCount\ ps' = 0)$
using *assms* **proof** (*induction badNtsCount ps arbitrary*: *ps rule*: *less_induct*)
case *less*
then show *?case*
proof (*cases badNtsCount ps = 0*)
case *False*
from *this* **obtain** $ps'\ A\ B_1\ B_2$ **where** *g'*: *binarizeNt A B1 B2 S ps ps'*
using *assms lemma8_b less*(*2*) **by** *blast*
hence *badNtsCount ps' < badNtsCount ps*
using *lemma6_b* **by** *blast*
from *this* **obtain** *ps''* **where** $(\lambda x\ y.\ \exists A\ B_1\ B_2.\ binarizeNt\ A\ B_1\ B_2\ S\ x\ y)^{**}\ ps'\ ps'' \wedge badNtsCount\ ps'' = 0$
using *less slemma15_a*[*of A B1 B2 S ps ps'*] *g'* **by** *blast*
then show *?thesis*
using *g' converse_rtranclp_into_rtranclp*[*of* $(\lambda x\ y.\ \exists A\ B_1\ B_2.\ binarizeNt\ A\ B_1\ B_2\ S\ x\ y)\ ps\ ps'\ ps''$] **by** *blast*
qed *blast*
qed

theorem *cnf_noe_nou*: **fixes** *ps* :: (*'n*::*infinite*,*'t*)*prods*
assumes *Eps_free* (*set ps*) **and** *Unit_free* (*set ps*)
shows $\exists ps'$::(*'n*,*'t*)*prods*. *uniform* (*set ps'*) $\wedge$ *binary* (*set ps'*) $\wedge$ *lang ps S* = *lang ps' S* $\wedge$ *Eps_free* (*set ps'*) $\wedge$ *Unit_free* (*set ps'*)
proof −
obtain *ps'* **where** *g'*: $(\lambda x\ y.\ \exists A\ t.\ uniformize\ A\ t\ S\ x\ y)^{**}\ ps\ ps' \wedge badTmsCount\ ps' = 0 \wedge Eps\_free\ (set\ ps') \wedge Unit\_free\ (set\ ps')$
using *assms uniformize_2 uniformizeRtc_Eps_free uniformizeRtc_Unit_free* **by** *blast*
obtain *ps''* **where** *g''*: $(\lambda x\ y.\ \exists A\ B_1\ B_2.\ binarizeNt\ A\ B_1\ B_2\ S\ x\ y)^{**}\ ps'\ ps'' \wedge (badNtsCount\ ps'' = 0) \wedge (badTmsCount\ ps'' = 0)$
using *g' binarizeNt_2 lemma15_a* **by** *blast*
hence *uniform* (*set ps''*) $\wedge$ *binary* (*set ps''*) $\wedge$ *Eps_free* (*set ps''*) $\wedge$ *Unit_free* (*set ps''*)
using *g' count_bin_un binarizeNtRtc_Eps_free binarizeNtRtc_Unit_free* **by** *fastforce*
moreover have *lang ps S* = *lang ps'' S*
using *g' g'' cnf_lemma* **by** *blast*
ultimately show *?thesis* **by** *blast*
qed

Alternative form more similar to the one Jana Hofmann used:

lemma *CNF_eq*: *CNF P* $\longleftrightarrow$ (*uniform P* $\wedge$ *binary P* $\wedge$ *Eps_free P* $\wedge$ *Unit_free P*)
proof

assume *CNF P*
hence *Eps_free P*
unfolding *CNF_def Eps_free_def* **by** *fastforce*
moreover have *Unit_free P*
using ‹*CNF P*› **unfolding** *CNF_def Unit_free_def isNt_def isTm_def* **by** *fastforce*
moreover have *uniform P*
proof −
have ∀ (*A*,*α*) ∈ *P*. (∃ *B C*. *α* = [*Nt B*, *Nt C*]) ∨ (∃ *t*. *α* = [*Tm t*])
using ‹*CNF P*› **unfolding** *CNF_def*.
hence ∀ (*A*, *α*) ∈ *P*. (∀ *N* ∈ *set α*. *isNt N*) ∨ (∃ *t*. *α* = [*Tm t*])
unfolding *isNt_def* **by** *fastforce*
hence ∀ (*A*, *α*) ∈ *P*. (∄ *t*. *Tm t* ∈ *set α*) ∨ (∃ *t*. *α* = [*Tm t*])
by (*auto simp*: *isNt_def*)
thus *uniform P*
unfolding *uniform_def*.
qed
moreover have *binary P*
using ‹*CNF P*› **unfolding** *binary_def CNF_def* **by** *auto*
ultimately show *uniform P* ∧ *binary P* ∧ *Eps_free P* ∧ *Unit_free P*
by *blast*
next
assume *assm*: *uniform P* ∧ *binary P* ∧ *Eps_free P* ∧ *Unit_free P*
have∀ *p* ∈ *P*. (∃ *B C*. (*snd p*) = [*Nt B*, *Nt C*]) ∨ (∃ *t*. (*snd p*) = [*Tm t*])
proof
fix *p* **assume** *p* ∈ *P*
obtain *A α* **where** *Aα*: (*A*, *α*) = *p*
by (*metis prod.exhaust_sel*)
hence *length α* = *1* ∨ *length α* = *2*
using *assm* ‹*p* ∈ *P*› *order_neq_le_trans* **unfolding** *binary_def Eps_free_def* **by** *fastforce*
hence (∃ *B C*. (*snd p*) = [*Nt B*, *Nt C*]) ∨ (∃ *t*. *α* = [*Tm t*])
proof
assume *length α* = *1*
hence ∃ *S*. *α* = [*S*]
by (*simp add*: *length_Suc_conv*)
moreover have ∄ *N*. *α* = [*Nt N*]
using *assm Aα* ‹*p* ∈ *P*› **unfolding** *Unit_free_def* **by** *blast*
ultimately show *?thesis* **by** (*metis sym.exhaust*)
next
assume *length α* = *2*
obtain *B C* **where** *BC*: *α* = [*B*, *C*]
using ‹*length α* = *2*› **by** (*metis One_nat_def Suc_1 diff_Suc_1 length_0_conv length_Cons neq_Nil_conv*)
have (∄ *t*. *Tm t* ∈ *set α*)
using ‹*length α* = *2*› *assm Aα* ‹*p* ∈ *P*› **unfolding** *uniform_def* **by** *auto*
hence (∀ *N* ∈ *set α*. ∃ *A*. *N* = *Nt A*)
by (*metis sym.exhaust*)
hence ∃ *B′ C′*. *B* = *Nt B′* ∧ *C* = *Nt C′*

using *BC* **by** *simp*
thus *?thesis* **using** *Aα BC* **by** *auto*
qed
thus (∃ *B C*. (*snd p*) = [*Nt B*, *Nt C*]) ∨ (∃ *t*. (*snd p*) = [*Tm t*]) **using** *Aα* **by** *auto*
qed
thus *CNF P* **by** (*auto simp*: *CNF_def*)
qed

Main Theorem: existence of CNF with the same language except for the empty word []:

theorem *cnf_exists*:
fixes *ps* :: ('*n*::*infinite*,'*t*) *prods*
shows ∃ *ps'*::('*n*,'*t*)*prods*. *CNF*(*set ps'*) ∧ *lang ps' S* = *lang ps S* − {[]}
proof −
obtain ps_0 **where** ps_0: *eps_elim_rel ps* ps_0
using *eps_elim_rel_exists* **by** *blast*
obtain ps_u **where** ps_u: *unit_elim_rel* ps_0 ps_u
using *unit_elim_rel_exists* **by** *blast*
hence *1*: *Eps_free* (*set* ps_u) ∧ *Unit_free* (*set* ps_u)
using ps_0 ps_u *Eps_free_if_eps_elim_rel Unit_free_if_unit_elim_rel unit_elim_rel_Eps_free* **by** *fastforce*
have *2*: *lang* ps_u *S* = *lang ps S* − {[]}
using ps_u *eps_elim_rel_lang_eq*[*OF* ps_0] *unit_elim_rel_lang_eq*[*OF* ps_u] **by** (*simp add*: *eps_elim_rel_lang_eq*)
obtain *ps'*::('*n*,'*t*)*prods* **where** *g'*: *uniform* (*set ps'*) ∧ *binary* (*set ps'*) ∧ *lang* ps_u *S* = *lang ps' S* ∧ *Eps_free* (*set ps'*) ∧ *Unit_free* (*set ps'*)
using *1 cnf_noe_nou* **by** *blast*
hence *CNF* (*set ps'*)
using *g' CNF_eq* **by** *blast*
moreover have *lang ps' S* = *lang ps S* − {[]}
using *2 g'* **by** *blast*
ultimately show *?thesis* **by** *blast*
qed

Some helpful properties:

lemma *cnf_length_derive*:
assumes *CNF P P* ⊢ [*Nt S*] ⇒∗ *α*
shows *length α* ≥ *1*
using *assms CNF_eq Eps_free_derives_Nil length_greater_0_conv less_eq_Suc_le* **by** *auto*

lemma *cnf_length_derive2*:
assumes *CNF P P* ⊢ [*Nt A*, *Nt B*] ⇒∗ *α*
shows *length α* ≥ *2*
proof −
obtain *u v* **where** *uv*: *P* ⊢ [*Nt A*] ⇒∗ *u* ∧ *P* ⊢ [*Nt B*] ⇒∗ *v* ∧ *α* = *u* @ *v*
using *assms*(*2*) *derives_append_decomp*[*of P* ‹[*Nt A*]› ‹[*Nt B*]› *α*] **by** *auto*
hence *length u* ≥ *1* ∧ *length v* ≥ *1*

using *cnf_length_derive*[*OF assms*(*1*)] **by** *blast*
thus *?thesis*
using *uv* **by** *simp*
qed

lemma *cnf_single_derive*:
assumes *CNF P P* ⊢ [*Nt S*] ⇒∗ [*Tm t*]
shows (*S*, [*Tm t*]) ∈ *P*
proof −
obtain α **where** α: *P* ⊢ [*Nt S*] ⇒ α ∧ *P* ⊢ α ⇒∗ [*Tm t*]
using *converse_rtranclpE*[*OF assms*(*2*)] **by** *auto*
hence *1*: (*S*, α) ∈ *P*
by (*simp add*: *derive_singleton*)
have ∄ *A B*. α = [*Nt A*, *Nt B*]
proof (*rule ccontr*)
assume ¬ (∄ *A B*. α = [*Nt A*, *Nt B*])
from *this* **obtain** *A B* **where** *AB*: α = [*Nt A*, *Nt B*]
by *blast*
have ∀ *w*. *P* ⊢ [*Nt A*, *Nt B*] ⇒∗ *w* ⟶ *length w* ≥ *2*
using *cnf_length_derive2*[*OF assms*(*1*)] **by** *simp*
moreover have *length* [*Tm t*] = *1*
by *simp*
ultimately show *False*
using α *AB* **by** *auto*
qed
from *this* **obtain** *a* **where** α = [*Tm a*]
using *1 assms*(*1*) **unfolding** *CNF_def* **by** *auto*
hence *t* = *a*
using α **by** (*simp add*: *derives_Tm_Cons*)
thus *?thesis*
using *1* ‹α = [*Tm a*]› **by** *blast*
qed

lemma *cnf_word*:
assumes *CNF P P* ⊢ [*Nt S*] ⇒∗ *map Tm w*
and *length w* ≥ *2*
shows ∃ *A B u v*. (*S*, [*Nt A*, *Nt B*]) ∈ *P* ∧ *P* ⊢ [*Nt A*] ⇒∗ *map Tm u* ∧ *P* ⊢ [*Nt B*] ⇒∗ *map Tm v* ∧ *u*@*v* = *w* ∧ *u* ≠ [] ∧ *v* ≠ []
proof −
have *1*: (*S*, *map Tm w*) ∉ *P*
using *assms*(*1*) *assms*(*3*) **unfolding** *CNF_def* **by** *auto*
have ∃ α. *P* ⊢ [*Nt S*] ⇒ α ∧ *P* ⊢ α ⇒∗ *map Tm w*
using *converse_rtranclpE*[*OF assms*(*2*)] **by** *auto*
from *this* **obtain** α **where** α: (*S*, α) ∈ *P* ∧ *P* ⊢ α ⇒∗ *map Tm w*
by (*auto simp*: *derive_singleton*)
hence (∄ *t*. α = [*Tm t*])
using *1 derives_Tm_Cons*[*of P*] *derives_from_empty* **by** *auto*
hence ∃ *A B*. *P* ⊢ [*Nt S*] ⇒ [*Nt A*, *Nt B*] ∧ *P* ⊢ [*Nt A*, *Nt B*] ⇒∗ *map Tm w*
using *assms*(*1*) α *derive_singleton*[*of P* ‹*Nt S*› α] **unfolding** *CNF_def* **by**

fast
from *this* **obtain** *A B* **where** *AB*: $(S, [Nt\ A, Nt\ B]) \in P \wedge P \vdash [Nt\ A, Nt\ B] \Rightarrow* map\ Tm\ w$
using *derive_singleton*[*of P* ‹*Nt S*›] **by** *blast*
hence $\neg(P \vdash [Nt\ A] \Rightarrow* []) \wedge \neg(P \vdash [Nt\ B] \Rightarrow* [])$
using *assms*(*1*) *CNF_eq Eps_free_derives_Nil* **by** *blast*
from *this* **obtain** *u v* **where** *uv*: $P \vdash [Nt\ A] \Rightarrow* u \wedge P \vdash [Nt\ B] \Rightarrow* v \wedge u@v = map\ Tm\ w \wedge u \neq [] \wedge v \neq []$
using *AB derives_append_decomp*[*of P* ‹[*Nt A*]› ‹[*Nt B*]› ‹*map Tm w*›] **by** *force*
moreover have $\exists u'\ v'.\ u = map\ Tm\ u' \wedge v = map\ Tm\ v'$
using *uv map_eq_append_conv*[*of Tm w u v*] **by** *auto*
ultimately show *?thesis*
using *AB append_eq_map_conv*[*of u v Tm w*] *list.simps*(*8*)[*of Tm*] **by** *fastforce*
qed

end

9 Pumping Lemma for Context Free Grammars

theory *Pumping_Lemma_CFG*
imports
List_Power.List_Power
Chomsky_Normal_Form
begin

Paths in the (implicit) parse tree of the derivation of some terminal word; specialized for productions in CNF.

inductive *path* :: $('n, 't)\ Prods \Rightarrow 'n \Rightarrow 'n\ list \Rightarrow 't\ list \Rightarrow bool$
$((2\_ \vdash / (\_ / \Rightarrow\langle\_\rangle / \_))\ [50, 0, 0, 50]\ 50)$ **where**
terminal: $(A, [Tm\ a]) \in P \Longrightarrow P \vdash A \Rightarrow\langle[A]\rangle\ [a]$ |
left: $[\![(A, [Nt\ B, Nt\ C]) \in P \wedge (P \vdash B \Rightarrow\langle p\rangle\ w) \wedge (P \vdash C \Rightarrow\langle q\rangle\ v)]\!] \Longrightarrow P \vdash A \Rightarrow\langle A\#p\rangle\ (w@v)$ |
right: $[\![(A, [Nt\ B, Nt\ C]) \in P \wedge (P \vdash B \Rightarrow\langle p\rangle\ w) \wedge (P \vdash C \Rightarrow\langle q\rangle\ v)]\!] \Longrightarrow P \vdash A \Rightarrow\langle A\#q\rangle\ (w@v)$

inductive *cnf_derives* :: $('n, 't)\ Prods \Rightarrow 'n \Rightarrow 't\ list \Rightarrow bool$
$((2\_ \vdash / (\_ / \Rightarrow cnf / \_))\ [50, 0, 50]\ 50)$ **where**
step: $(A, [Tm\ a]) \in P \Longrightarrow P \vdash A \Rightarrow cnf\ [a]$|
trans: $[\![(A, [Nt\ B, Nt\ C]) \in P \wedge P \vdash B \Rightarrow cnf\ w \wedge P \vdash C \Rightarrow cnf\ v]\!] \Longrightarrow P \vdash A \Rightarrow cnf\ (w@v)$

lemma *path_if_cnf_derives*: $P \vdash S \Rightarrow cnf\ w \Longrightarrow \exists p.\ P \vdash S \Rightarrow\langle p\rangle\ w$
by (*induction rule*: *cnf_derives.induct*) (*auto intro*: *path.intros*)

lemma *cnf_derives_if_path*: $P \vdash S \Rightarrow\langle p\rangle\ w \Longrightarrow P \vdash S \Rightarrow cnf\ w$
by (*induction rule*: *path.induct*) (*auto intro*: *cnf_derives.intros*)

corollary *cnf_path*: $P \vdash S \Rightarrow cnf\ w \longleftrightarrow (\exists\, p.\ P \vdash S \Rightarrow\langle p\rangle\ w)$
using *path_if_cnf_derives*[*of P*] *cnf_derives_if_path*[*of P*] **by** *blast*

lemma *cnf_der*:
assumes $P \vdash [Nt\ S] \Rightarrow* map\ Tm\ w\ CNF\ P$
shows $P \vdash S \Rightarrow cnf\ w$
using *assms* **proof** (*induction w arbitrary*: *S rule*: *length_induct*)
case (*1 w*)
have *Eps_free P*
using *assms*(*2*) *CNF_eq* **by** *blast*
hence $\neg(P \vdash [Nt\ S] \Rightarrow* [])$
by (*simp add*: *Eps_free_derives_Nil*)
hence *2*: *length w* $\neq$ *0*
using *1* **by** *auto*
thus *?case*
proof (*cases length w* $\leq$ *1*)
case *True*
hence *length w* = *1*
using *2* **by** *linarith*
then obtain *t* **where** *w* = [*t*]
using *length_Suc_conv*[*of w 0*] **by** *auto*
hence (*S*, [*Tm t*]) $\in$ *P*
using *1 assms*(*2*) *cnf_single_derive*[*of P S t*] **by** *simp*
thus *?thesis*
by (*simp add*: ‹*w* = _› *cnf_derives.intros*(*1*))
next
case *False*
obtain *A B u v* **where** *ABuv*: (*S*, [*Nt A*, *Nt B*]) $\in$ *P* $\wedge$ *P* $\vdash$ [*Nt A*] $\Rightarrow*$ *map Tm u* $\wedge$ *P* $\vdash$ [*Nt B*] $\Rightarrow*$ *map Tm v* $\wedge$ *u*@*v* = *w* $\wedge$ *u* $\neq$ [] $\wedge$ *v* $\neq$ []
using *False assms*(*2*) *1 cnf_word*[*of P S w*] **by** *auto*
have *length u* < *length w* $\wedge$ *length v* < *length w*
using *ABuv* **by** *auto*
hence *cnf_derives P A u* $\wedge$ *cnf_derives P B v*
using *1 ABuv* **by** *blast*
thus *?thesis*
using *ABuv cnf_derives.intros*(*2*)[*of S A B P u v*] **by** *blast*
qed
qed

lemma *der_cnf*:
assumes $P \vdash S \Rightarrow cnf\ w\ CNF\ P$
shows $P \vdash [Nt\ S] \Rightarrow* map\ Tm\ w$
using *assms* **proof** (*induction rule*: *cnf_derives.induct*)
case (*step A a P*)
then show *?case*
by (*simp add*: *derive_singleton r_into_rtranclp*)
next
case (*trans A B C P w v*)
hence $P \vdash [Nt\ A] \Rightarrow [Nt\ B, Nt\ C]$

by (*simp add*: *derive_singleton*)
moreover have $P \vdash [Nt\ B] \Rightarrow* map\ Tm\ w \wedge P \vdash [Nt\ C] \Rightarrow* map\ Tm\ v$
using *trans* **by** *blast*
ultimately show *?case*
using *derives_append_decomp*[*of P* ‹[*Nt B*]› ‹[*Nt C*]› ‹*map Tm* (*w* @ *v*)›] **by** *auto*
qed

corollary *cnf_der_eq*: *CNF P* $\Longrightarrow$ ($P \vdash [Nt\ S] \Rightarrow* map\ Tm\ w \longleftrightarrow P \vdash S \Rightarrow cnf\ w$)
using *cnf_der*[*of P S w*] *der_cnf*[*of P S w*] **by** *blast*

lemma *path_if_derives*:
assumes $P \vdash [Nt\ S] \Rightarrow* map\ Tm\ w\ CNF\ P$
shows $\exists p.\ P \vdash S \Rightarrow\langle p\rangle\ w$
using *assms cnf_der*[*of P S w*] *path_if_cnf_derives*[*of P S w*] **by** *blast*

lemma *derives_if_path*:
assumes $P \vdash S \Rightarrow\langle p\rangle\ w\ CNF\ P$
shows $P \vdash [Nt\ S] \Rightarrow* map\ Tm\ w$
using *assms der_cnf*[*of P S w*] *cnf_derives_if_path*[*of P S p w*] **by** *blast*

lpath = longest path, similar to *path*; *lpath* always chooses the longest path in a syntax tree

inductive *lpath* :: ('*n*, '*t*) *Prods* $\Rightarrow$ '*n* $\Rightarrow$ '*n list* $\Rightarrow$ '*t list* $\Rightarrow$ *bool*
((*2*_ $\vdash$/ (_/ $\Rightarrow\langle$_$\rangle$/ _)) [*50*, *0*, *0*, *50*] *50*) **where**
terminal: $(A, [Tm\ a]) \in P \Longrightarrow P \vdash A \Rightarrow\langle\!\langle [A]\rangle\!\rangle\ [a]$ |
nonTerminals: $[\![(A, [Nt\ B, Nt\ C]) \in P \wedge (P \vdash B \Rightarrow\langle\!\langle p\rangle\!\rangle\ w) \wedge (P \vdash C \Rightarrow\langle\!\langle q\rangle\!\rangle\ v)]\!] \Longrightarrow$
$P \vdash A \Rightarrow\langle\!\langle A\#(if\ length\ p \geq length\ q\ then\ p\ else\ q)\rangle\!\rangle\ (w@v)$

lemma *path_lpath*: $P \vdash S \Rightarrow\langle p\rangle\ w \Longrightarrow \exists p'.\ (P \vdash S \Rightarrow\langle\!\langle p'\rangle\!\rangle\ w) \wedge length\ p' \geq length\ p$
by (*induction rule*: *path.induct*) (*auto intro*: *lpath.intros*)

lemma *lpath_path*: $(P \vdash S \Rightarrow\langle\!\langle p\rangle\!\rangle\ w) \Longrightarrow P \vdash S \Rightarrow\langle p\rangle\ w$
by (*induction rule*: *lpath.induct*) (*auto intro*: *path.intros*)

lemma *lpath_length*: $(P \vdash S \Rightarrow\langle\!\langle p\rangle\!\rangle\ w) \Longrightarrow length\ w \leq 2\hat{}length\ p$
proof (*induction rule*: *lpath.induct*)
case (*terminal A a P*)
then show *?case* **by** *simp*
next
case (*nonTerminals A B C P p w q v*)
then show *?case*
proof (*cases length p* $\geq$ *length q*)
case *True*
hence *length v* $\leq$ *2^length p*

using *nonTerminals order_subst1*[*of* ‹*length v*› ‹λ*x*. 2^*x*› ‹*length q*› ‹*length p*›] **by** *simp*
hence *length w* + *length v* ≤ 2∗2^*length p*
by (*simp add*: *nonTerminals.IH*(1) *add_le_mono mult_2*)
then show *?thesis*
by (*simp add*: *True*)
next
case *False*
hence *length w* ≤ 2^*length q*
using *nonTerminals order_subst1*[*of* ‹*length w*› ‹λ*x*. 2^*x*› ‹*length p*› ‹*length q*›] **by** *simp*
hence *length w* + *length v* ≤ 2∗2^*length q*
by (*simp add*: *nonTerminals.IH add_le_mono mult_2*)
then show *?thesis*
by (*simp add*: *False*)
qed
qed

lemma *step_decomp*:
assumes $P \vdash A \Rightarrow\langle [A]@p\rangle\ w$ *length p* ≥ 1
shows $\exists B\ C\ p'\ a\ b.\ (A, [Nt\ B, Nt\ C]) \in P \wedge w = a@b \wedge$
$((P \vdash B \Rightarrow\langle p\rangle\ a \wedge P \vdash C \Rightarrow\langle p'\rangle\ b) \vee (P \vdash B \Rightarrow\langle p'\rangle\ a \wedge P \vdash C \Rightarrow\langle p\rangle\ b))$
using *assms* **by** (*cases*) *fastforce*+

lemma *steplp_decomp*:
assumes $P \vdash A \Rightarrow\langle\!\langle [A]@p\rangle\!\rangle\ w$ *length p* ≥ 1
shows $\exists B\ C\ p'\ a\ b.\ (A, [Nt\ B, Nt\ C]) \in P \wedge w = a@b \wedge$
$((P \vdash B \Rightarrow\langle\!\langle p\rangle\!\rangle\ a \wedge P \vdash C \Rightarrow\langle\!\langle p'\rangle\!\rangle\ b \wedge$ *length p* ≥ *length p'*) ∨
$(P \vdash B \Rightarrow\langle\!\langle p'\rangle\!\rangle\ a \wedge P \vdash C \Rightarrow\langle\!\langle p\rangle\!\rangle\ b \wedge$ *length p* ≥ *length p'*))
using *assms* **by** (*cases*) *fastforce*+

lemma *path_first_step*: $P \vdash A \Rightarrow\langle p\rangle\ w \Longrightarrow \exists q.\ p = [A]@q$
by (*induction rule*: *path.induct*) *simp_all*

lemma *no_empty*: $P \vdash A \Rightarrow\langle p\rangle\ w \Longrightarrow$ *length w* > 0
by (*induction rule*: *path.induct*) *simp_all*

lemma *substitution*:
assumes $P \vdash A \Rightarrow\langle p1@[X]@p2\rangle\ z$
shows $\exists v\ w\ x.\ P \vdash X \Rightarrow\langle [X]@p2\rangle\ w \wedge z = v@w@x \wedge$
$(\forall w'\ p'.\ P \vdash X \Rightarrow\langle [X]@p'\rangle\ w' \longrightarrow P \vdash A \Rightarrow\langle p1@[X]@p'\rangle\ v@w'@x) \wedge$
(*length p1* > 0 ⟶ *length* (*v*@*x*) > 0)
using *assms* **proof** (*induction p1 arbitrary*: *P A z*)
case *Nil*
hence $\forall w'\ p'.\ ((P \vdash X \Rightarrow\langle [X]@p2\rangle\ z) \wedge z = []@z@[] \wedge$
$(P \vdash X \Rightarrow\langle [X]@p'\rangle\ w' \longrightarrow P \vdash A \Rightarrow\langle []@[X]@p'\rangle\ ([]@w'@[])) \wedge$
(*length* [] > 0 ⟶ *length* ([]@[]) > 0))
using *path_first_step*[*of P A*] **by** *auto*
then show *?case*

by *blast*
next
case (*Cons A p1 P Y*)
hence *0*: $A = Y$
using *path_first_step*[*of P Y*] **by** *fastforce*
have *length* $(p1@[X]@p2) \geq 1$
by *simp*
hence $\exists B\ C\ a\ b\ q.\ (A, [Nt\ B, Nt\ C]) \in P \wedge a@b = z \wedge$
$((P \vdash B \Rightarrow\langle q\rangle\ a \wedge P \vdash C \Rightarrow\langle p1@[X]@p2\rangle\ b) \vee (P \vdash B \Rightarrow\langle p1@[X]@p2\rangle\ a \wedge P \vdash C \Rightarrow\langle q\rangle\ b))$
using *Cons.prems path_first_step step_decomp* **by** *fastforce*
then obtain *B C a b q* **where** *BC*: $(A, [Nt\ B, Nt\ C]) \in P \wedge a@b = z \wedge$
$((P \vdash B \Rightarrow\langle q\rangle\ a \wedge P \vdash C \Rightarrow\langle p1@[X]@p2\rangle\ b) \vee (P \vdash B \Rightarrow\langle p1@[X]@p2\rangle\ a \wedge P \vdash C \Rightarrow\langle q\rangle\ b))$
by *blast*
then show *?case*
proof (*cases* $P \vdash B \Rightarrow\langle q\rangle\ a \wedge P \vdash C \Rightarrow\langle p1@[X]@p2\rangle\ b$)
case *True*
then obtain *v w x* **where** *vwx*: $P \vdash X \Rightarrow\langle[X]@p2\rangle\ w \wedge b = v@w@x \wedge$
$(\forall w'\ p'.\ P \vdash X \Rightarrow\langle[X]@p'\rangle\ w' \longrightarrow P \vdash C \Rightarrow\langle p1@[X]@p'\rangle\ (v@w'@x))$
using *Cons.IH* **by** *blast*
hence *1*: $\forall w'\ p'.\ (P \vdash X \Rightarrow\langle[X]@p'\rangle\ w') \longrightarrow P \vdash A \Rightarrow\langle A\#p1@[X]@p'\rangle\ (a@v@w'@x)$
using *BC* **by** (*auto intro*: *path.intros*(*3*))
obtain v' **where** $v' = a@v$
by *simp*
hence *length* $(v'@x) > 0$
using *True no_empty* **by** *fast*
hence $P \vdash X \Rightarrow\langle[X]@p2\rangle\ w \wedge z = v'@w@x \wedge (\forall w'\ p'.$
$P \vdash X \Rightarrow\langle[X]@p'\rangle\ w' \longrightarrow P \vdash A \Rightarrow\langle A\#p1@[X]@p'\rangle\ (v'@w'@x)) \wedge$
$(length\ (A\#p1) > 0 \longrightarrow length\ (v'@x) > 0)$
using *vwx 1 BC* ‹$v' = \_$› **by** *simp*
thus *?thesis*
using *0* **by** *auto*
next
case *False*
then obtain *v w x* **where** *vwx*: $(P \vdash X \Rightarrow\langle[X]@p2\rangle\ w) \wedge a = v@w@x \wedge$
$(\forall w'\ p'.\ P \vdash X \Rightarrow\langle[X]@p'\rangle\ w' \longrightarrow P \vdash B \Rightarrow\langle p1@[X]@p'\rangle\ (v@w'@x))$
using *Cons.IH BC* **by** *blast*
hence *1*: $\forall w'\ p'.\ P \vdash X \Rightarrow\langle[X]@p'\rangle\ w' \longrightarrow P \vdash A \Rightarrow\langle A\#p1@[X]@p'\rangle\ (v@w'@x@b)$
using *BC left*[*of A B C P*] **by** *fastforce*
hence $P \vdash X \Rightarrow\langle[X]@p2\rangle\ w \wedge z = v@w@x@b \wedge (\forall w'\ p'.$
$P \vdash X \Rightarrow\langle[X]@p'\rangle\ w' \longrightarrow P \vdash A \Rightarrow\langle A\#p1@[X]@p'\rangle\ (v@w'@x@b)) \wedge$
$(length\ (A\#p1) > 0 \longrightarrow length\ (a@v@x) > 0)$
using *vwx BC no_empty* **by** *fastforce*
moreover have *length* $(v@x@b) > 0$
using *no_empty BC* **by** *fast*
ultimately show *?thesis*
using *0* **by** *auto*

qed
qed

lemma *substitution_lp*:
assumes $P \vdash A \Rightarrow\langle\!\langle p1@[X]@p2\rangle\!\rangle\ z$
shows $\exists\, v\ w\ x.\ P \vdash X \Rightarrow\langle\!\langle [X]@p2\rangle\!\rangle\ w \wedge z = v@w@x \wedge$
$(\forall\, w'\ p'.\ P \vdash X \Rightarrow\langle [X]@p'\rangle\ w' \longrightarrow P \vdash A \Rightarrow\langle p1@[X]@p'\rangle\ v@w'@x)$
using *assms* **proof** (*induction p1 arbitrary*: *P A z*)
case *Nil*
hence $\forall\, w'\ p'.\ P \vdash X \Rightarrow\langle [X]@p'\rangle\ w' \longrightarrow P \vdash A \Rightarrow\langle []@[X]@p'\rangle\ ([]@w'@[])$
using *path_first_step lpath_path* **by** *fastforce*
moreover have $P \vdash X \Rightarrow\langle\!\langle [X]@p2\rangle\!\rangle\ z \wedge z = []@z@[]$
using *Nil lpath.cases*[*of P A* ‹[*X*] @ *p2*› *z*] **by** *auto*
ultimately show *?case*
by *blast*
next
case (*Cons A p1 P Y*)
hence *0*: $A = Y$
using *path_first_step*[*of P Y*] *lpath_path* **by** *fastforce*
have *length* $(p1@[X]@p2) \geq 1$
by *simp*
hence $\exists\, B\ C\ p'\ a\ b.\ (A, [Nt\ B, Nt\ C]) \in P \wedge z = a@b \wedge$
$((P \vdash B \Rightarrow\langle\!\langle p1@[X]@p2\rangle\!\rangle\ a \wedge P \vdash C \Rightarrow\langle\!\langle p'\rangle\!\rangle\ b \wedge length\ (p1@[X]@p2) \geq length\ p') \vee$
$(P \vdash B \Rightarrow\langle\!\langle p'\rangle\!\rangle\ a \wedge P \vdash C \Rightarrow\langle\!\langle p1@[X]@p2\rangle\!\rangle\ b \wedge length\ (p1@[X]@p2) \geq length\ p'))$
using *steplp_decomp*[*of P A* ‹*p1*@[*X*]@*p2*› *z*] *0 Cons* **by** *simp*
then obtain *B C p' a b* **where** *BC*: $(A, [Nt\ B, Nt\ C]) \in P \wedge z = a@b \wedge$
$((P \vdash B \Rightarrow\langle\!\langle p1@[X]@p2\rangle\!\rangle\ a \wedge P \vdash C \Rightarrow\langle\!\langle p'\rangle\!\rangle\ b \wedge length\ (p1@[X]@p2) \geq length\ p') \vee$
$(P \vdash B \Rightarrow\langle\!\langle p'\rangle\!\rangle\ a \wedge P \vdash C \Rightarrow\langle\!\langle p1@[X]@p2\rangle\!\rangle\ b \wedge length\ (p1@[X]@p2) \geq length\ p'))$
by *blast*
then show *?case*
proof (*cases* $P \vdash B \Rightarrow\langle\!\langle p1@[X]@p2\rangle\!\rangle\ a \wedge P \vdash C \Rightarrow\langle\!\langle p'\rangle\!\rangle\ b \wedge length\ (p1@[X]@p2) \geq length\ p'$)
case *True*
then obtain *v w x* **where** *vwx*: $P \vdash X \Rightarrow\langle\!\langle [X]@p2\rangle\!\rangle\ w \wedge a = v@w@x \wedge$
$(\forall\, w'\ p'.\ P \vdash X \Rightarrow\langle [X]@p'\rangle\ w' \longrightarrow P \vdash B \Rightarrow\langle p1@[X]@p'\rangle\ v@w'@x)$
using *Cons.IH* **by** *blast*
hence $P \vdash X \Rightarrow\langle\!\langle [X]@p2\rangle\!\rangle\ w \wedge z = v@w@x@b \wedge$
$(\forall\, w'\ p'.\ P \vdash X \Rightarrow\langle [X]@p'\rangle\ w' \longrightarrow P \vdash A \Rightarrow\langle A\#p1@[X]@p'\rangle\ v@w'@x@b)$
using *BC lpath_path*[*of P*] *path.intros*(*2*)[*of A B C P*] **by** *fastforce*
then show *?thesis*
using *0* **by** *auto*
next
case *False*
hence $(P \vdash B \Rightarrow\langle\!\langle p'\rangle\!\rangle\ a \wedge P \vdash C \Rightarrow\langle\!\langle p1@[X]@p2\rangle\!\rangle\ b \wedge length\ (p1@[X]@p2) \geq length\ p')$

using *BC* **by** *blast*
then obtain *v w x* **where** *vwx*: $P \vdash X \Rightarrow\langle\!\langle [X]@p2 \rangle\!\rangle\ w \wedge b = v@w@x \wedge$
$(\forall\, w'\ p'.\ P \vdash X \Rightarrow\langle [X]@p' \rangle\ w' \longrightarrow P \vdash C \Rightarrow\langle p1@[X]@p' \rangle\ v@w'@x)$
using *Cons.IH* **by** *blast*
then obtain *v'* **where** $v' = a@v$
by *simp*
hence $P \vdash X \Rightarrow\langle\!\langle [X]@p2 \rangle\!\rangle\ w \wedge z = v'@w@x \wedge$
$(\forall\, w'\ p'.\ P \vdash X \Rightarrow\langle [X]@p' \rangle\ w' \longrightarrow P \vdash A \Rightarrow\langle A\#p1@[X]@p' \rangle\ v'@w'@x)$
using *BC lpath_path*[*of P*] *path.intros*(*3*)[*of A B C P*] *vwx* **by** *fastforce*
then show *?thesis*
using *0* **by** *auto*
qed
qed

lemma *path_nts*: $P \vdash S \Rightarrow\langle p \rangle\ w \Longrightarrow set\ p \subseteq Nts\ P$
unfolding *Nts_def* **by** (*induction rule*: *path.induct*) *auto*

lemma *inner_aux*:
assumes $\forall\, w'\ p'.\ P \vdash A \Rightarrow\langle [A]@p3 \rangle\ w \wedge (P \vdash A \Rightarrow\langle [A]@p' \rangle\ w' \longrightarrow$
$P \vdash A \Rightarrow\langle [A]@p2@[A]@p' \rangle\ v@w'@x)$
shows $P \vdash A \Rightarrow\langle (([A]@p2)^{\frown\frown}(Suc\ i)) @ [A]@p3 \rangle\ v^{\frown\frown}(Suc\ i) @ w @ x^{\frown\frown}(Suc\ i)$
using *assms* **proof** (*induction i*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc i*)
hence *1*: $P \vdash A \Rightarrow\langle [A]@p2@ ([A] @ p2)^{\frown\frown}i @ [A]@p3 \rangle\ v^{\frown\frown}(Suc\ i) @ w @ x^{\frown\frown}(Suc\ i)$
by *simp*
hence $P \vdash A \Rightarrow\langle [A] @ p2 @ [A] @ p2@ ([A] @ p2)^{\frown\frown}i @ [A]@p3 \rangle\ v @ v^{\frown\frown}(Suc\ i) @ w @ x^{\frown\frown}(Suc\ i) @ x$
using *assms* **by** *fastforce*
thus *?case*
using *pow_list_Suc2*[*of* ‹*Suc i*› *x*] **by** *simp*
qed

lemma *inner_pumping*:
assumes *CNF P finite P*
and $m = card\ (Nts\ P)$
and $z \in Lang\ P\ S$
and $length\ z \geq 2^{\frown}(m+1)$
shows $\exists\, u\ v\ w\ x\ y\ .\ z = u@v@w@x@y \wedge length\ (v@w@x) \leq 2^{\frown}(m+1) \wedge length\ (v@x) \geq 1 \wedge (\forall\, i.\ u@(v^{\frown\frown}i)@w@(x^{\frown\frown}i)@y \in Lang\ P\ S)$
proof −
obtain *p'* **where** *p'*: $P \vdash S \Rightarrow\langle p' \rangle\ z$
using *assms Lang_def*[*of P S*] *path_if_derives*[*of P S*] **by** *blast*
then obtain *lp* **where** *lp*: $P \vdash S \Rightarrow\langle\!\langle lp \rangle\!\rangle\ z$
using *path_lpath*[*of P*] **by** *blast*
hence *1*: $set\ lp \subseteq Nts\ P$

```
    using lpath_path[of P] path_nts[of P] by blast
  have length lp > m
  proof −
    have (2^(m+1)::nat) ≤ 2^length lp
      using lp lpath_length[of P S lp z] assms(5) le_trans by blast
    hence m+1 ≤ length lp
      using power_le_imp_le_exp[of 2 ‹m+1› ‹length lp›] by auto
    thus ?thesis
      by simp
  qed
  then obtain l p where p: lp = l@p ∧ length p = m+1
    using less_Suc_eq by (induction lp) fastforce+
  hence set l ⊆ Nts P ∧ set p ⊆ Nts P ∧ finite (Nts P)
    using ‹finite P› 1 finite_Nts[of P] assms(1) by auto
  hence card (set p) < length p
    using p assms(3) card_mono[of ‹Nts P› ‹set p›] by simp
  then obtain A p1 p2 p3 where p = p1@[A]@p2@[A]@p3
    by (metis distinct_card nat_neq_iff not_distinct_decomp)
   then obtain u vwx y where uy: (P ⊢ A ⇒⟪[A]@p2@[A]@p3⟫ vwx ∧ z =
u@vwx@y ∧
        (∀ w' p'. (P ⊢ A ⇒⟨[A]@p'⟩ w' ⟶ P ⊢ S ⇒⟨l@p1@[A]@p'⟩ u@w'@y)))
    using substitution_lp[of P S ‹l@p1› A ‹p2@[A]@p3› z] lp p by auto
  hence length vwx ≤ 2^(m+1)
   using ‹p = _› p lpath_length[of P A ‹[A] @ p2 @ [A] @ p3› vwx] order_subst1
by fastforce
  then obtain v w x where vwx: P ⊢ A ⇒⟨[A]@p3⟩ w ∧ vwx = v@w@x ∧
       (∀ w' p'. (P ⊢ A ⇒⟨[A]@p'⟩ w' ⟶ P ⊢ A ⇒⟨[A]@p2@[A]@p'⟩ v@w'@x)) ∧
       (length ([A]@p2) > 0 ⟶ length (v@x) > 0)
    using substitution[of P A ‹[A]@p2› A p3 vwx] uy lpath_path[of P A] by auto
  have P ⊢ S ⇒⟨l@p1@ (([A]@p2)^^i) @[A]@p3⟩ u@(v^^i)@w@(x^^i)@y for i
  proof −
    have ∀ i. P ⊢ A ⇒⟨([A]@p2)^^(Suc i) @ [A]@p3⟩ v^^(Suc i) @ w @ x^^(Suc
i)
      using vwx inner_aux[of P A] by blast
    hence ∀ i. P ⊢ S ⇒⟨l@p1@(([A]@p2)^^(Suc i)) @[A]@p3⟩ u@ (v^^(Suc i)) @
w @ (x^^(Suc i)) @y
      using uy by fastforce
    moreover have P ⊢ S ⇒⟨l@p1@(([A]@p2)^^0) @[A]@p3⟩ u@ (v^^0) @ w @
(x^^0) @y
      using vwx uy by auto
   ultimately show P ⊢ S ⇒⟨l@p1@ (([A]@p2)^^i) @[A]@p3⟩ u@(v^^i)@w@(x^^i)@y
      by (induction i) simp_all
  qed
  hence ∀ i. u@(v^^i)@w@(x^^i)@y ∈ Lang P S
     unfolding Lang_def using assms(1) assms(2) derives_if_path[of P S] by
blast
  hence z = u@v@w@x@y ∧ length (v@w@x) ≤ 2^(m+1) ∧ 1 ≤ length (v@x) ∧
(∀ i. u@(v^^i)@w@(x^^i)@ y ∈ Lang P S)
    using vwx uy ‹length vwx ≤ 2 ^ (m + 1)› by (simp add: Suc_leI)
```

then show *?thesis*
by *blast*
qed

abbreviation *pumping_property L n* ≡ ∀ *z* ∈ *L*. *length z* ≥ *n* ⟶
(∃ *u v w x y*. *z* = *u* @ *v* @ *w* @ *x* @ *y* ∧ *length* (*v*@*w*@*x*) ≤ *n* ∧ *length* (*v*@*x*) ≥ *1*
∧ (∀ *i*. *u* @ *v*^^*i* @ *w* @ *x*^^*i* @ *y* ∈ *L*))

theorem *Pumping_Lemma_CNF*:
assumes *CNF P finite P*
shows ∃ *n*. *pumping_property* (*Lang P S*) *n*
using *inner_pumping*[*OF assms*, *of* ‹*card* (*Nts P*)›] **by** *blast*

theorem *Pumping_Lemma*:
assumes *finite* (*P* :: (*'n*::*infinite*,*'t*)*Prods*)
shows ∃ *n*. *pumping_property* (*Lang P S*) *n*
proof −
obtain *ps* **where** *set ps* = *P* **using** *finite_list*[*OF assms*] **by** *blast*
obtain *ps'* :: (*'n*,*'t*)*prods* **where** *ps'*: *CNF*(*set ps'*) *lang ps' S*= *lang ps S* − {[]}
using *cnf_exists*[*of S ps*] **by** *auto*
let *?P'* = *set ps'*
have *P'*: *CNF ?P' finite ?P'* **using** *ps'*(*1*) **by** *auto*
from *Pumping_Lemma_CNF*[*OF P'*, *of S*] **obtain** *n* **where**
pump: *pumping_property* (*Lang ?P' S*) *n* **by** *blast*
then have *pumping_property* (*Lang ?P' S*) (*Suc n*)
by (*metis Suc_leD nle_le*)
then have *pumping_property* (*Lang P S*) (*Suc n*)
using *ps'*(*2*) ‹*set ps* = *P*› **by** (*metis Diff_iff list.size*(*3*) *not_less_eq_eq singletonD zero_le*)
then show *?thesis* **by** *blast*
qed

end

10 $a^n b^n c^n$ is Not Context-Free

theory *AnBnCn_not_CFL*
imports *Pumping_Lemma_CFG*
begin

This theory proves that the language $\bigcup n\ \{[a]^n\ @\ [b]^n\ @\ [c]^n\}$ is not context-free using the Pumping lemma.

The formal proof follows the textbook proof (e.g. [1]) closely. The Isabelle proof is about 10% of the length of the Coq proof by Ramos *et al.* [3, 2]. The latter proof suffers from excessive case analyses.

declare *count_list_pow_list*[*simp*]

context
fixes *a b c*
assumes *neq*: *a≠b b≠c c≠a*
begin

lemma *c_greater_count0*:
assumes *x@y = [a]^^n @ [b]^^n @ [c]^^n length y≥n*
shows *count_list x c = 0*
using *assms* **proof** −
have *drop (2∗n) (x@y) = [c]^^n* **using** *assms*
by *simp*
then have *count_c_end*: *count_list (drop (2∗n) (x@y)) c = n*
by (*simp*)
have *count_list (x@y) c= n* **using** *assms neq*
by (*simp*)
then have *count_c_front*: *count_list (take (2∗n) (x@y)) c = 0*
using *count_c_end* **by** (*metis add_cancel_left_left append_take_drop_id count_list_append*)
have ∃ *i*. *length y = n+i* **using** *assms*
by *presburger*
then obtain *i* **where** *i*: *length y= n+i*
by *blast*
then have *x = take (3∗n−n−i) (x@y)*
proof −
have *x = take (2 ∗ n − i) x @ take (2 ∗ n − (i + length x)) y*
using *i* **by** (*metis add.commute add_diff_cancel_left′ append_eq_conv_conj assms(1) diff_diff_left*
length_append length_pow_list_single mult_2 take_append)
then show *?thesis*
by (*simp*)
qed
then have *x = take (3∗n−n−i) (take (3∗n−n) (x@y))*
by (*metis diff_le_self min_def take_take*)
then have *x = take (3∗n−n−i) (take (2∗n) (x@y))*
by *fastforce*
then show *?thesis* **using** *count_c_front count_list_0_iff in_set_takeD*
by *metis*
qed

lemma *a_greater_count0*:
assumes *x@y = [a]^^n @ [b]^^n @ [c]^^n length x≥n*
shows *count_list y a = 0*

this prof is easier than ⟦*?x @ ?y = [a]^?n @ [b]^?n @ [c]^?n*; *?n ≤ length ?y*⟧ ⟹ *count_list ?x c = 0* since a is at the start of the word rather than at the end

proof −
have *count_whole*: *count_list (x@y) a = n*
using *assms neq* **by** *auto*

```
  have take_n: take n (x@y) = [a]^^n
    using assms by simp
  then have count_take_n: count_list (take n (x@y)) a = n
    by (simp)
  have ∃ z. x = take n (x@y) @ z
    by (metis append_eq_conv_conj assms(2) nat_le_iff_add take_add)
 then have  count_a_x:count_list x a = n using count_take_n take_n count_whole

   by (metis add_diff_cancel_left' append.right_neutral count_list_append diff_add_zero)
  have count_list (x@y) a = n
    using assms neq by simp
   then have count_list y a = 0
    using count_a_x by simp
  then show ?thesis
    by presburger
qed

lemma a_or_b_zero:
  assumes u@w@y = [a]^^n @ [b]^^n @ [c]^^n length w ≤ n
  shows count_list w a = 0 ∨ count_list w c = 0
```

This lemma uses *count_list w a = 0 ∨ count_list w c = 0* similar to all following proofs, focusing on the number of a and c found in w rather than the concrete structure. It is also the merge of the two previous lemmas to make the final proof shorter

```
proof−
  show ?thesis proof (cases length u <n)
    case True
    have length (u@w@y) = 3∗n  using assms
      by simp
    then have length u + length w + length y = 3∗n
      by simp
    then have length u + length y ≥ 2∗n using assms
      by linarith
    then have u_or_y: length u ≥ n ∨ length y ≥ n
      by linarith
    then  have length y≥n using True
      by simp
    then show ?thesis using c_greater_count0[of u@w y n] neq assms
      by simp
  next
    case False
    then have length u ≥ n
      by simp
    then show ?thesis using a_greater_count0[of u w@y n ] neq assms
      by auto
  qed
qed
```

lemma *count_vx_not_zero*:
assumes $u@v@w@x@y = [a]^{\frown\frown}n\ @\ [b]^{\frown\frown}n\ @\ [c]^{\frown\frown}n\ v@x \neq []$
shows *count_list* $(v@x)\ a \neq 0 \vee$ *count_list* $(v@x)\ b \neq 0 \vee$ *count_list* $(v@x)\ c \neq 0$
proof −
have *set*: *set* $([a]^{\frown\frown}n\ @\ [b]^{\frown\frown}n\ @\ [c]^{\frown\frown}n) = \{a,b,c\}$ **using** *assms pow_list_single_Nil_iff*
by (*fastforce simp add*: *pow_list_single*)
show *?thesis* **proof** (*cases* $v \neq []$)
case *True*
then have $\exists\, d \in set([a]^{\frown\frown}n\ @\ [b]^{\frown\frown}n\ @\ [c]^{\frown\frown}n).$ *count_list v d* $\neq 0$
using *assms*(*1*)
by (*metis append_Cons count_list_0_iff in_set_conv_decomp list.exhaust list.set_intros*(*1*))
then have *count_list v a* $\neq 0 \vee$ *count_list v b* $\neq 0 \vee$ *count_list v c* $\neq 0$
using *set* **by** *simp*
then show *?thesis*
by *force*
next
case *False*
then have $x \neq []$ **using** *assms*
by *fast*
then have $\exists\, d \in set\ ([a]^{\frown\frown}n\ @\ [b]^{\frown\frown}n\ @\ [c]^{\frown\frown}n).$ *count_list x d* $\neq 0$
proof −
have $\exists\, d \in set\ ([a]\ ^{\frown\frown}\ n) \cup (set\ ([b]\ ^{\frown\frown}\ n) \cup set\ ([c]\ ^{\frown\frown}\ n)).\ ya \neq d \longrightarrow 0 <$ *count_list ys d*
if $u\ @\ v\ @\ w\ @\ ya\ \#\ ys\ @\ y = [a]\ ^{\frown\frown}\ n\ @\ [b]\ ^{\frown\frown}\ n\ @\ [c]\ ^{\frown\frown}\ n$
and $x = ya\ \#\ ys$
for *ya* :: *'a*
and *ys* :: *'a list*
using *that* **by** (*metis Un_iff in_set_conv_decomp set_append*)
then show *?thesis*
using *assms*(*1*) ‹$x \neq []$› **by** (*auto simp*: *neq_Nil_conv*)
qed
then have *count_list x a* $\neq 0 \vee$ *count_list x b* $\neq 0 \vee$ *count_list x c* $\neq 0$
using *set* **by** *simp*
then show *?thesis*
by *force*
qed
qed

lemma *not_ex_y_count*:
assumes $i \neq k \vee k \neq j \vee i \neq j$ *count_list w a* $= i$ *count_list w b* $= k$ *count_list w c* $= j$
shows $\neg(EX\ y.\ w = [a]^{\frown\frown}y\ @\ [b]^{\frown\frown}y\ @\ [c]^{\frown\frown}y)$
proof
assume *EX y*. $w = [a]^{\frown\frown}y\ @\ [b]^{\frown\frown}y\ @\ [c]^{\frown\frown}y$
then obtain *y* **where** *y*: $w = [a]^{\frown\frown}y\ @\ [b]^{\frown\frown}y\ @\ [c]^{\frown\frown}y$
by *blast*
then have *count_list w a* $= y$ **using** *neq*

```
    by simp
  then have i_eq_y: i=y using assms
    by argo
  then have count_list w b = y
    using neq assms(2) y by (auto)
  then have k_eq_y: k=y using assms
    by argo
  have count_list w c = y using neq y
    by simp
  then have j_eq_y: j=y using assms
    by argo
  show False  using i_eq_y k_eq_y j_eq_y assms
     by presburger
qed

lemma not_in_count:
  assumes  count_list w a ≠ count_list w b ∨ count_list w b ≠ count_list w c ∨
count_list w c ≠ count_list w a
  shows w ∉ {word. ∃ n.  word = [a]^^n @ [b]^^n @ [c]^^n}
```

This definition of a word not in the language is useful as it allows us to prove a word is not in the language just by knowing the number of each letter in a word

```
  using assms not_ex_y_count
  by (smt (verit, del_insts) mem_Collect_eq)
```

This is the central and only case analysis, which follows the textbook proofs. The Coq proof by Ramos is considerably more involved and ends up with a total of 24 cases

```
lemma pumping_application:
  assumes u@v@w@x@y = [a]^^n @ [b]^^n @ [c]^^n
  and count_list (v@w@x) a = 0 ∨ count_list (v@w@x) c = 0 and v@x≠[]
  shows u@w@y ∉ (⋃ n. {[a]^^n @ [b]^^n @ [c]^^n})
```

In this lemma it is proven that a word $u \mathbin{@} v^0 \mathbin{@} w \mathbin{@} x^0 \mathbin{@} y$ is not in the language $\bigcup_n \{[a]^n \mathbin{@} [b]^n \mathbin{@} [c]^n\}$ as this is the easiest counterexample useful for the Pumping lemma

```
proof−
  have count_word_a: count_list (u@v@w@x@y) a = n
    using neq assms(1) by simp
  have count_word_b: count_list (u@v@w@x@y) b = n
    using neq assms(1) by simp
  have count_word_c: count_list (u@v@w@x@y) c = n
    using neq assms(1) by simp
  have count_non_zero: ((count_list (v@x) a) ≠0) ∨ (count_list (v@x) b≠0) ∨
(count_list (v@x) c ≠ 0)
    using count_vx_not_zero[of u v w x y n] assms(1,3) by simp
  from assms(2) show ?thesis proof
    assume ∗: count_list (v @ w @ x) a = 0
```

then have *vx_b_or_c_not0*: *count_list* (*v*@*x*) *b* ≠ *0* ∨ *count_list* (*v*@*x*) *c* ≠ *0* **using** *count_non_zero*
by *simp*
have *uwy_count_a*: *count_list* (*u*@*w*@*y*) *a* = *n* **using** ∗ *count_word_a*
by *simp*
have *count_list* (*u*@*w*@*y*) *b* ≠ *n* ∨ *count_list* (*u*@*w*@*y*) *c* ≠ *n* **using** *vx_b_or_c_not0 count_word_b count_word_c*
by *auto*
then have (*count_list* (*u*@*w*@*y*) *a* ≠ *count_list* (*u*@*w*@*y*) *b*) ∨ (*count_list* (*u*@*w*@*y*) *c* ≠ *count_list* (*u*@*w*@*y*) *a*) **using** *uwy_count_a*
by *argo*
then show *?thesis* **using** *not_in_count*[*of u*@*w*@*y*]
by *blast*
next
assume ∗: *count_list* (*v* @ *w* @ *x*) *c* = *0*
then have *vx_a_or_b_not0*: (*count_list* (*v*@*x*) *a*≠*0*) ∨ (*count_list* (*v*@*x*) *b* ≠ *0*) **using** *count_non_zero*
by *fastforce*
have *uwy_count_c*: *count_list* (*u*@*w*@*y*) *c*=*n* **using** ∗ *count_word_c*
by *auto*
have *count_list* (*u*@*w*@*y*) *a* ≠*n* ∨ *count_list* (*u*@*w*@*y*) *b* ≠*n* **using** *vx_a_or_b_not0 count_word_a count_word_b*
by *force*
then have (*count_list* (*u*@*w*@*y*) *a* ≠ *count_list* (*u*@*w*@*y*) *b*) ∨ (*count_list* (*u*@*w*@*y*) *c* ≠ *count_list* (*u*@*w*@*y*) *a*) **using** *uwy_count_c*
by *argo*
then show *?thesis* **using** *not_in_count*[*of u*@*w*@*y*]
by *blast*
qed
qed

theorem *anbncn_not_cfl*:
assumes *finite* (*P* :: (*'n*::*infinite*,*'a*)*Prods*)
shows *Lang P S* ≠ (⋃ *n*. {[*a*]^^*n* @ [*b*]^^*n* @ [*c*]^^*n*}) (**is** ¬ *?E*)
proof
assume *?E*
from *Pumping_Lemma*[*OF* ‹*finite P*›, *of S*] **obtain** *n* **where**
pump: ∀ *word* ∈ *Lang P S*. *length word* ≥ *n* ⟶
(∃ *u v w x y*. *word* = *u*@*v*@*w*@*x*@*y* ∧ *length* (*v*@*w*@*x*) ≤ *n* ∧ *length* (*v*@*x*) ≥ *1* ∧ (∀ *i*. *u*@(*v*^^*i*)@*w*@(*x*^^*i*)@*y* ∈ *Lang P S*))
by *blast*
let *?word* = [*a*]^^*n* @ [*b*]^^*n* @ [*c*]^^*n*
have *wInLg*: *?word* ∈ *Lang P S*
using ‹*?E*› **by** *blast*
have *length ?word* ≥ *n*
by *simp*
then obtain *u v w x y* **where** *uvwxy*: *?word* = *u*@*v*@*w*@*x*@*y* ∧ *length* (*v*@*w*@*x*) ≤ *n* ∧ *length* (*v*@*x*) ≥ *1* ∧ (∀ *i*. *u*@(*v*^^*i*)@*w*@(*x*^^*i*)@*y* ∈ *Lang P S*)
using *pump wInLg* **by** *metis*

let *?vwx*= *v*@*w*@*x*
have (*count_list ?vwx a=0*) ∨ (*count_list ?vwx c=0*) **using** *uvwxy a_or_b_zero assms*
by (*metis* (*no_types*, *lifting*) *append.assoc*)
then show *False* **using** *assms uvwxy pumping_application*[*of u v w x y n*]
by (*metis* ‹*?E*› *append_Nil length_0_conv not_one_le_zero pow_list.simps*(*1*))
qed

end

end

11 CFLs Are Not Closed Under Intersection

theory *CFL_Not_Intersection_Closed*
imports
List_Power.List_Power
Context_Free_Language
Pumping_Lemma_CFG
AnBnCn_not_CFL
begin

The probably first formal proof was given by Ramos *et al.* [3, 2]. The proof below is much shorter.

Some lemmas:

lemma *Lang_concat*:
L1 @@ *L2* = {*word*. ∃ *w1* ∈ *L1*. ∃ *w2* ∈ *L2*. *word* = *w1* @ *w2*}
unfolding *conc_def* **by** *blast*

lemma *deriven_same_repl*:
assumes (*A*, *u'* @ [*Nt A*] @ *v'*) ∈ *P*
shows *P* ⊢ *u* @ [*Nt A*] @ *v* ⇒(*n*) *u* @ (*u'*^^*n*) @ [*Nt A*] @ (*v'*^^*n*) @ *v*
proof (*induction n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
have *P* ⊢ *u* @ (*u'*^^*n*) @ [*Nt A*] @ (*v'*^^*n*) @ *v* ⇒ *u* @ (*u'*^^*n*) @ *u'* @ [*Nt A*] @ *v'* @ (*v'*^^*n*) @ *v*
using *assms derive.intros*[*of* _ _ _ *u* @ (*u'*^^*n*) (*v'*^^*n*) @ *v*] **by** *fastforce*
then have *P* ⊢ *u* @ (*u'*^^*n*) @ [*Nt A*] @ (*v'*^^*n*) @ *v* ⇒ *u* @ (*u'*^^(*Suc n*)) @ [*Nt A*] @ (*v'*^^(*Suc n*)) @ *v*
by (*metis append.assoc pow_list_Suc2 pow_list_comm*)
then show *?case* **using** *Suc* **by** *auto*
qed

Now the main proof.

lemma *an_CFL*: *CFL TYPE*('*n*) (⋃ *n*. {[*a*]^^*n*}) (**is** *CFL* _ *?L*)

proof −
obtain P X **where** P_def: $(P::('n, 'a)\ Prods) = \{(X, [Tm\ a, Nt\ X]), (X, [])\}$ **by** *simp*
have $Lang\ P\ X = ?L$
proof
show $Lang\ P\ X \subseteq ?L$
proof
fix w
assume $w \in Lang\ P\ X$
then have $P \vdash [Nt\ X] \Rightarrow* map\ Tm\ w$ **using** *Lang_def* **by** *fastforce*
then have $\exists n.\ map\ Tm\ w = ([Tm\ a]^\frown\frown n)\ @\ [Nt\ X] \vee (map\ Tm\ w::('n, 'a)syms) = ([Tm\ a]^\frown\frown n)$
proof(*induction rule*: *derives_induct*)
case *base*
then show *?case* **by** (*auto simp*: *pow_list_single_Nil_iff*)
next
case (*step u A v w'*)
then have $A{=}X$ **using** *P_def* **by** *auto*
have $P \vdash u\ @\ [Nt\ X]\ @\ v \Rightarrow u\ @\ w'\ @\ v$ **using** ‹$A{=}X$› *derive.intros step* **by** *fast*
obtain n **where** n_src: $u\ @\ [Nt\ X]\ @\ v = ([Tm\ a]^\frown\frown n)\ @\ [Nt\ X] \vee u\ @\ [Nt\ X]\ @\ v = ([Tm\ a]^\frown\frown n)$
using ‹$A{=}X$› *step* **by** *auto*
have *notin*: $Nt\ X \notin set\ ([Tm\ a]^\frown\frown n)$ **by** (*simp add*: *pow_list_single*)
then have $u\ @\ [Nt\ X]\ @\ v = ([Tm\ a]^\frown\frown n)\ @\ [Nt\ X]$
using *n_src append_Cons in_set_conv_decomp* **by** *metis*
then have *uv_eq*: $u = ([Tm\ a]^\frown\frown n) \wedge v = []$
using *notin n_src Cons_eq_appendI append_Cons_eq_iff append_Nil in_set_insert insert_Nil snoc_eq_iff_butlast* **by** *metis*
have $w' = [Tm\ a, Nt\ X] \vee w' = []$ **using** *step*(*2*) *P_def* **by** *auto*
then show *?case*
proof
assume $w' = [Tm\ a, Nt\ X]$
then have $u\ @\ w'\ @\ v = ([Tm\ a]^\frown(Suc\ n))\ @\ [Nt\ X]$ **using** *uv_eq* **by** (*simp add*: *pow_list_comm*)
then show *?case* **by** *blast*
next
assume $w' = []$
then show *?case* **using** *uv_eq* **by** *blast*
qed
qed
then obtain n' **where** n'_src: $(map\ Tm\ w) = ([Tm\ a]^\frown\frown n')\ @\ [Nt\ X] \vee ((map\ Tm\ w)::('n, 'a)syms) = ([Tm\ a]^\frown\frown n')$ **by** *auto*
have $Nt\ X \notin set\ (map\ Tm\ w)$ **by** *auto*
then have $((map\ Tm\ w)::('n, 'a)syms) = ([Tm\ a]^\frown\frown n')$ **using** n'_src **by** *fastforce*
have $map\ Tm\ ([a]^\frown\frown n') = ([Tm\ a]^\frown\frown n')$ **by** (*simp add*: *map_concat*)
then have $w = [a]^\frown\frown n'$ **using** ‹$map\ Tm\ w = ([Tm\ a]^\frown\frown n')$› **by** (*metis list.inj_map_strong sym.inject*(*2*))

```
    then show w ∈ ?L by auto
   qed
  next
   show ?L ⊆ Lang P X
   proof
    fix w
    assume w ∈ ?L
    then obtain n where n_src: w = [a]⌢n by blast

    have Xa: P ⊢ [Nt X] ⇒(n) ([Tm a]⌢n) @ [Nt X]
      using P_def deriven_same_repl[of X [Tm a] [] _ _ [] ] by (simp add:
pow_list_Nil)
     have Xε: P ⊢ ([Tm a]⌢n) @ [Nt X] ⇒ ([Tm a]⌢n) using P_def de-
rive.intros[of X [] _ [Tm a]⌢n []] by auto
    have ([Tm a]⌢n) = map Tm w using n_src by auto
    then have P ⊢ [Nt X] ⇒* map Tm w using Xa Xε relpowp_imp_rtranclp
     by (smt (verit, best) rtranclp.simps)
    then show w ∈ Lang P X using Lang_def by fastforce
   qed
  qed
  then show ?thesis unfolding CFL_def P_def by blast
qed

lemma anbn_CFL: CFL TYPE('n) (⋃n. {[a]⌢n @ [b]⌢n}) (is CFL _ ?L)
proof −
  obtain P X where P_def: (P::('n, 'a) Prods) = {(X, [Tm a, Nt X, Tm b]), (X,
[])} by simp
  let ?G = Cfg P X
  have Lang P X = ?L
  proof
   show Lang P X ⊆ ?L proof
    fix w
    assume w ∈ Lang P X
    then have P ⊢ [Nt X] ⇒* map Tm w using Lang_def by fastforce
    then have ∃n. map Tm w = ([Tm a]⌢n) @ [Nt X] @ ([Tm b]⌢n) ∨ (map
Tm w::('n, 'a)syms) = ([Tm a]⌢n) @ ([Tm b]⌢n)
    proof(induction rule: derives_induct)
     case base
     have [Nt X] = ([Tm a]⌢0) @ [Nt X] @ ([Tm b]⌢0) by auto
     then show ?case by fast
    next
     case (step u A v w')
     then have A=X using P_def by auto
      have P ⊢ u @ [Nt X] @ v ⇒ u @ w' @ v using ‹A=X› derive.intros step
by fast
     obtain n where n_src: u @ [Nt X] @ v = ([Tm a]⌢n) @ [Nt X] @ ([Tm
b]⌢n) ∨ u @ [Nt X] @ v = ([Tm a]⌢n) @ ([Tm b]⌢n)
      using ‹A=X› step by auto
     have notin2: Nt X ∉ set ([Tm a]⌢n) ∧ Nt X ∉ set ([Tm b]⌢n)
```

```
      by (simp add: pow_list_single)
    then have notin: Nt X ∉ set (([Tm a]⁀⁀n) @ ([Tm b]⁀⁀n)) by simp
     then have uv_split: u @ [Nt X] @ v = ([Tm a]⁀⁀n) @ [Nt X] @ ([Tm
b]⁀⁀n)
      by (metis n_src append_Cons in_set_conv_decomp)
    have u_eq: u = ([Tm a]⁀⁀n)
        by (metis (no_types, lifting) uv_split notin2 Cons_eq_appendI ap-
pend_Cons_eq_iff eq_Nil_appendI)
    then have v_eq: v = ([Tm b]⁀⁀n)
        by (metis (no_types, lifting) uv_split notin2 Cons_eq_appendI ap-
pend_Cons_eq_iff eq_Nil_appendI)
    have w' = [Tm a, Nt X, Tm b] ∨ w' = [] using step(2) P_def by auto
    then show ?case
    proof
     assume w' = [Tm a, Nt X, Tm b]
      then have u @ w' @ v = ([Tm a]⁀⁀n) @ [Tm a, Nt X, Tm b] @ ([Tm
b]⁀⁀n) using u_eq v_eq by simp
       then have u @ w' @ v = ([Tm a]⁀⁀(Suc n)) @ [Nt X] @ ([Tm b]⁀⁀(Suc
n))
        by (simp add: pow_list_comm)
     then show ?case by blast
    next
     assume w' = []
     then show ?case using u_eq v_eq by blast
    qed
  qed
 then obtain n' where n'_src: (map Tm w) = ([Tm a]⁀⁀n') @ [Nt X] @ ([Tm
b]⁀⁀n') ∨ ((map Tm w)::('n, 'a)syms) = ([Tm a]⁀⁀n') @ ([Tm b]⁀⁀n') by auto
  have Nt X ∉ set (map Tm w) by auto
 then have w_ab: ((map Tm w)::('n, 'a)syms) = ([Tm a]⁀⁀n') @ ([Tm b]⁀⁀n')
using n'_src by fastforce
  have map_a: ([Tm a]⁀⁀n') = map Tm ([a]⁀⁀n') by (simp add: map_concat)
  have map_b: ([Tm b]⁀⁀n') = map Tm ([b]⁀⁀n') by (simp add: map_concat)
   from w_ab map_a map_b have ((map Tm w)::('n, 'a)syms) = map Tm
([a]⁀⁀n') @  map Tm ([b]⁀⁀n') by metis
 then have ((map Tm w)::('n, 'a)syms) = map Tm (([a]⁀⁀n') @ ([b]⁀⁀n')) by
simp
 then have w = [a]⁀⁀n' @ [b]⁀⁀n' by (metis list.inj_map_strong sym.inject(2))
  then show w ∈ ?L by auto
 qed
next
 show ?L ⊆ Lang P X
 proof
  fix w
  assume w ∈ ?L
  then obtain n where n_src: w = [a]⁀⁀n @ [b]⁀⁀n by blast

  have Xa: P ⊢ [Nt X] ⇒(n) ([Tm a]⁀⁀n) @ [Nt X] @ ([Tm b]⁀⁀n)
   using P_def deriven_same_repl[of X [Tm a] [Tm b] _ _ [] []] by simp
```

have $X\varepsilon$: $P \vdash ([Tm\ a]@@n)\ @\ [Nt\ X]\ @\ ([Tm\ b]@@n) \Rightarrow ([Tm\ a]@@n)\ @\ ([Tm\ b]@@n)$
using *P_def derive.intros*[*of X* [] _ $[Tm\ a]@@n$ $[Tm\ b]@@n$] **by** *auto*
have $[Tm\ a]@@n\ @\ [Tm\ b]@@n = map\ Tm\ ([a]@@n)\ @\ (map\ Tm\ ([b]@@n))$ **by** *simp*
then have $([Tm\ a]@@n)\ @\ ([Tm\ b]@@n) = map\ Tm\ w$ **using** *n_src* **by** *simp*
then have $P \vdash [Nt\ X] \Rightarrow* map\ Tm\ w$ **using** *Xa Xε relpowp_imp_rtranclp*
by (*smt* (*verit*, *best*) *rtranclp.simps*)
then show $w \in Lang\ P\ X$ **using** *Lang_def* **by** *fastforce*
qed
qed
then show *?thesis* **unfolding** *CFL_def P_def* **by** *blast*
qed

lemma *intersection_anbncn*:
assumes $a{\neq}b\ b{\neq}c\ c{\neq}a$
and $\exists x\ y\ z.\ w = [a]@@x@[b]@@y@[c]@@z \wedge x = y$
and $\exists x\ y\ z.\ w = [a]@@x@[b]@@y@[c]@@z \wedge y = z$
shows $\exists x\ y\ z.\ w = [a]@@x@[b]@@y@[c]@@z \wedge x = y \wedge y = z$
proof −
obtain *x1 y1 z1* **where** *src1*: $w = [a]@@x1@[b]@@y1@[c]@@z1 \wedge x1 = y1$ **using** *assms* **by** *blast*
obtain *x2 y2 z2* **where** *src2*: $w = [a]@@x2@[b]@@y2@[c]@@z2 \wedge y2 = z2$ **using** *assms* **by** *blast*
have $[a]@@x1@[b]@@y1@[c]@@z1 = [a]@@x2@[b]@@y2@[c]@@z2$ **using** *src1 src2* **by** *simp*
have *cx1*: *count_list w a* = *x1* **using** *src1 assms* **by** (*simp*)
have *cx2*: *count_list w a* = *x2* **using** *src2 assms* **by** (*simp*)
from *cx1 cx2* **have** *eqx*: *x1* = *x2* **by** *simp*

have *cy1*: *count_list w b* = *y1* **using** *assms src1* **by** (*simp*)
have *cy2*: *count_list w b* = *y2* **using** *assms src2* **by** *simp*
from *cy1 cy2* **have** *eqy*: *y1* = *y2* **by** *simp*

have *cz1*: *count_list w c* = *z1* **using** *assms src1* **by** *simp*
have *cz2*: *count_list w c* = *z2* **using** *assms src2* **by** *simp*
from *cz1 cz2* **have** *eqz*: *z1* = *z2* **by** *simp*

have $w = [a]@@x1@[b]@@y1@[c]@@z1 \wedge x1 = y1 \wedge y1 = z1$ **using** *eqx eqy eqz src1 src2* **by** *blast*
then show *?thesis* **by** *blast*
qed

lemma *CFL_intersection_not_closed*:
fixes *a b c* :: *'a*
assumes $a{\neq}b\ b{\neq}c\ c{\neq}a$
shows $\exists L1\ L2 ::$ *'a list set*.
CFL TYPE((*'n1* + *'n1*) *option*) *L1* $\wedge$ *CFL TYPE*((*'n2* + *'n2*) *option*) *L2*
$\wedge$ ($\nexists$(*P*::(*'x*::*infinite*,*'a*)*Prods*) *S*. *Lang P S* = *L1* $\cap$ *L2* $\wedge$ *finite P*)

proof −
let *?anbn* = $\bigcup n.\ \{[a]^\frown n\ @\ [b]^\frown n\}$
let *?cm* = $\bigcup n.\ \{[c]^\frown n\}$
let *?an* = $\bigcup n.\ \{[a]^\frown n\}$
let *?bmcm* = $\bigcup n.\ \{[b]^\frown n\ @\ [c]^\frown n\}$
let *?anbncm* = $\bigcup n.\ \bigcup m.\ \{[a]^\frown n\ @\ [b]^\frown n\ @\ [c]^\frown m\}$
let *?anbmcm* = $\bigcup n.\ \bigcup m.\ \{[a]^\frown n\ @\ [b]^\frown m\ @\ [c]^\frown m\}$
have *anbn*: *CFL TYPE*('*n1*) *?anbn* **by**(*rule anbn_CFL*)
have *cm*: *CFL TYPE*('*n1*) *?cm* **by**(*rule an_CFL*)
have *an*: *CFL TYPE*('*n2*) *?an* **by**(*rule an_CFL*)
have *bmcm*: *CFL TYPE*('*n2*) *?bmcm* **by**(*rule anbn_CFL*)
have *?anbncm* = *?anbn @@ ?cm* **unfolding** *Lang_concat* **by** *auto*
then have *anbncm*: *CFL TYPE*((*'n1* + *'n1*) *option*) *?anbncm* **using** *anbn cm CFL_concat_closed* **by** *auto*
have *?anbmcm* = *?an @@ ?bmcm* **unfolding** *Lang_concat* **by** *auto*
then have *anbmcm*: *CFL TYPE*((*'n2* + *'n2*) *option*) *?anbmcm* **using** *an bmcm CFL_concat_closed* **by** *auto*
have *?anbncm* ∩ *?anbmcm* = $(\bigcup\ n.\ \{[a]^\frown n@[b]^\frown n@[c]^\frown n\})$
using *intersection_anbncn*[*OF assms*] **by** *auto*
then have *CFL TYPE*((*'n1* + *'n1*) *option*) *?anbncm* ∧
CFL TYPE((*'n2* + *'n2*) *option*) *?anbmcm* ∧
(∄ *P*::(*'x*,*'a*)*Prods*.∃ *S*. *Lang P S* = *?anbncm* ∩ *?anbmcm* ∧ *finite P*)
using *anbncn_not_cfl*[*OF assms*, **where** *'n* = *'x*] *anbncm anbmcm* **by** *auto*
then show *?thesis* **by** *auto*
qed

end

12 Inlining a Single Production

theory *Inlining1Prod*
imports *Context_Free_Grammar*
begin

A single production of (A,α) can be removed from *ps* by inlining (= replacing *Nt A* by α everywhere in *ps*) without changing the language if *Nt A* ∉ *set* α and $A \notin$ *lhss ps*.

substP ps s u replaces every occurrence of the symbol *s* with the list of symbols *u* on the right-hand sides of the production list *ps*

definition *substP* :: ('*n*, '*t*) *sym* ⇒ ('*n*, '*t*) *syms* ⇒ ('*n*, '*t*) *prods* ⇒ ('*n*, '*t*) *prods*
where
substP s u ps = *map* (λ(*A*,*v*). (*A*, *substs s u v*)) *ps*

lemma *substP_split*: *substP s u* (*ps* @ *ps'*) = *substP s u ps* @ *substP s u ps'*
by (*simp add*: *substP_def*)

lemma *substP_skip1*: *s* ∉ *set v* ⟹ *substP s u* ((*A*,*v*) # *ps*) = (*A*,*v*) # *substP s u ps*

by (*auto simp add*: *substs_skip substP_def*)

lemma *substP_skip2*: $s \notin syms\ ps \Longrightarrow substP\ s\ u\ ps = ps$
by (*induction ps*) (*auto simp add*: *substP_def substs_skip*)

lemma *substP_rev*: $Nt\ B \notin syms\ ps \Longrightarrow substP\ (Nt\ B)\ [s]\ (substP\ s\ [Nt\ B]\ ps) = ps$
proof (*induction ps*)
case *Nil*
then show *?case*
by (*simp add*: *substP_def*)
next
case (*Cons a ps*)
let *?A* = *fst a* **let** *?u* = *snd a* **let** *?substs* = *substs s* [*Nt B*]
have *substP* (*Nt B*) [*s*] (*substP s* [*Nt B*] (*a* # *ps*)) = *substP* (*Nt B*) [*s*] (*map* (λ(*A*,*v*). (*A*, *?substs v*)) (*a*#*ps*))
by (*simp add*: *substP_def*)
also have ... = *substP* (*Nt B*) [*s*] ((*?A*, *?substs ?u*) # *map* (λ(*A*,*v*). (*A*, *?substs v*)) *ps*)
by (*simp add*: *case_prod_unfold*)
also have ... = *map* ((λ(*A*,*v*). (*A*, *substsNt B* [*s*] *v*))) ((*?A*, *?substs ?u*) # *map* (λ(*A*,*v*). (*A*, *?substs v*)) *ps*)
by (*simp add*: *substP_def*)
also have ... = (*?A*, *substsNt B* [*s*] (*?substs ?u*)) # *substP* (*Nt B*) [*s*] (*substP s* [*Nt B*] *ps*)
by (*simp add*: *substP_def*)
also from *Cons* **have** ... = (*?A*, *substsNt B* [*s*] (*?substs ?u*)) # *ps*
using *set_subset_Cons* **unfolding** *Syms_def* **by** *auto*
also from *Cons.prems* **have** ... = (*?A*, *?u*) # *ps*
using *substs_rev* **unfolding** *Syms_def* **by** *force*
also have ... = *a* # *ps* **by** *simp*
finally show *?case* **.**
qed

lemma *substP_der*:
(*A*,*u*) ∈ *set* (*substP* (*Nt B*) *v ps*) ⟹ *set* ((*B*,*v*) # *ps*) ⊢ [*Nt A*] ⇒∗ *u*
proof −
assume (*A*,*u*) ∈ *set* (*substP* (*Nt B*) *v ps*)
then have ∃ *u′*. (*A*,*u′*) ∈ *set ps* ∧ *u* = *substsNt B v u′* **unfolding** *substP_def* **by** *auto*
then obtain *u′* **where** *u′_def*: (*A*,*u′*) ∈ *set ps* ∧ *u* = *substsNt B v u′* **by** *blast*
then have *path1*: *set* ((*B*,*v*) # *ps*) ⊢ [*Nt A*] ⇒∗ *u′*
by (*simp add*: *derive_singleton r_into_rtranclp*)
have *set* ((*B*,*v*) # *ps*) ⊢ *u′* ⇒∗ *substsNt B v u′*
using *substs_der* **by** (*metis list.set_intros*(*1*))
with *u′_def* **have** *path2*: *set* ((*B*,*v*) # *ps*) ⊢ *u′* ⇒∗ *u* **by** *simp*
from *path1 path2* **show** *?thesis* **by** *simp*
qed

A list of symbols u can be derived before inlining if u can be derived

after inlining.

lemma *if_part*:
  set (*substP* (*Nt B*) *v ps*) ⊢ [*Nt A*] ⇒∗ *u* ⟹ *set* ((*B*,*v*) # *ps*) ⊢ [*Nt A*] ⇒∗ *u*
proof (*induction rule*: *derives_induct*)
  case (*step u A v w*)
  then show *?case*
    by (*meson derives_append derives_prepend rtranclp_trans substP_der*)
qed *simp*

For the other implication we need to take care that B can be derived in the original *ps*. Thus, after inlining, B must also be substituted in the derived sentence *u*:

lemma *only_if_lemma*:
assumes *A* ≠ *B*
    and *B* ∉ *lhss ps*
    and *Nt B* ∉ *set v*
shows *set* ((*B*,*v*)#*ps*) ⊢ [*Nt A*] ⇒∗ *u* ⟹ *set* (*substP* (*Nt B*) *v ps*) ⊢ [*Nt A*] ⇒∗ *substsNt B v u*
proof (*induction rule*: *derives_induct*)
  case *base*
  then show *?case* **using** *assms*(*1*) **by** *simp*
next
  case (*step s B′ w v′*)
  then show *?case*
  proof (*cases B* = *B′*)
    case *True*
    then have *v* = *v′*
      using ‹*B* ∉ *lhss ps*› *step.hyps* **unfolding** *Lhss_def* **by** *auto*
    then have *substsNt B v* (*s* @ [*Nt B′*] @ *w*) = *substsNt B v* (*s* @ *v′* @ *w*)
      using *step.prems* ‹*Nt B* ∉ *set v*› *True* **by** (*simp add*: *substs_skip*)
    then show *?thesis*
      using *step.IH* **by** *argo*
  next
    case *False*
    then have (*B′*,*v′*) ∈ *set ps*
      using *step* **by** *auto*
    then have (*B′*, *substsNt B v v′*) ∈ *set* (*substP* (*Nt B*) *v ps*)
      by (*metis* (*no_types*, *lifting*) *list.set_map pair_imageI substP_def*)
    from *derives_rule*[*OF this _ rtranclp.rtrancl_refl*] *step.IH False* **show** *?thesis*
      by *simp*
  qed
qed

With the assumption that the non-terminal B is not in the list of symbols *u*, *substs u* (*Nt B*) *v* reduces to *u*

corollary *only_if_part*:
assumes *A* ≠ *B*
    and *B* ∉ *lhss ps*
    and *Nt B* ∉ *set v*

and $Nt\ B \notin set\ u$
shows $set\ ((B,v)\#ps) \vdash [Nt\ A] \Rightarrow* u \Longrightarrow set\ (substP\ (Nt\ B)\ v\ ps) \vdash [Nt\ A] \Rightarrow* u$
by (*metis assms substs_skip only_if_lemma*)

Combining the two implications gives us the desired property of language preservation

lemma *derives_inlining*:
assumes $B \notin lhss\ ps$ **and**
$Nt\ B \notin set\ v$ **and**
$Nt\ B \notin set\ u$ **and**
$A \neq B$
shows $set\ (substP\ (Nt\ B)\ v\ ps) \vdash [Nt\ A] \Rightarrow* u \longleftrightarrow set\ ((B,v)\ \#\ ps) \vdash [Nt\ A] \Rightarrow* u$
using *assms if_part only_if_part* **by** *metis*

end

13 Transforming Long Productions Into a Binary Cascade

theory *Binarize*
imports *Inlining1Prod*
begin

lemma *funpow_fix*: **fixes** $f :: 'a \Rightarrow 'a$ **and** $m :: 'a \Rightarrow nat$
assumes $\bigwedge x.\ m(f\ x) < m\ x \vee f\ x = x$
shows $f((f\ \widehat{}\widehat{}\ m\ x)\ x) = (f\ \widehat{}\widehat{}\ m\ x)\ x$
proof −
have $n + m\ ((f\widehat{}\widehat{}n)\ x) \leq m\ x \vee f((f\widehat{}\widehat{}n)\ x) = (f\widehat{}\widehat{}n)\ x$ **for** n
proof(*induction n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
then show *?case*
proof
assume *a1*: $n + m\ ((f\ \widehat{}\widehat{}\ n)\ x) \leq m\ x$
then show *?thesis*
proof (*cases* $m\ ((f\ \widehat{}\widehat{}\ Suc\ n)\ x) < m\ ((f\ \widehat{}\widehat{}\ n)\ x)$)
case *True*
then show *?thesis* **using** *a1* **by** *auto*
next
case *False*
with *assms* **have** $(f\ \widehat{}\widehat{}\ Suc\ n)\ x = (f\ \widehat{}\widehat{}\ n)\ x$ **by** *auto*
then show *?thesis* **by** *simp*
qed
next

```
    assume f ((f ^^ n) x) = (f ^^ n) x
    then show ?thesis by simp
   qed
 qed
  from this[of m x] show ?thesis using assms[of (f ^^ m x) x] by auto
qed
```

In a binary grammar, every right-hand side consists of at most two symbols. The *binarize* function should convert an arbitrary production list into a binary production list, without changing the language of the grammar. For this we make use of fixpoint iteration and define the function *binarize1* for splitting a production, whose right-hand side exceeds the maximum number of symbols 2, into two productions. The step function is then defined as the auxiliary function *binarize′*. We also define the count function *count* that counts the right-hand sides whose length is more than or equal to 2

```
fun binarize1 :: ('n :: fresh0, 't) prods ⇒ ('n, 't) prods ⇒ ('n, 't) prods where
  binarize1 ps' [] = []
| binarize1 ps' ((A, []) # ps) = (A, []) # binarize1 ps' ps
| binarize1 ps' ((A, s0 # u) # ps) =
 (if length u ≤ 1 then (A, s0 # u) # binarize1 ps' ps
  else let B = fresh0 (nts ps') in (A,[s0, Nt B]) # (B, u) # ps)

definition binarize' :: ('n::fresh0, 't) prods ⇒ ('n, 't) prods where
  binarize' ps = binarize1 ps ps

fun count :: ('n, 't) prods ⇒ nat where
  count [] = 0
| count ((A,u) # ps) = (if length u ≤ 2 then count ps else length u + count ps)

definition binarize :: ('n::fresh0, 't) prods ⇒ ('n, 't) prods where
  binarize ps = (binarize' ^^ (count ps)) ps


lemma binarize [(0::nat, [Tm (0::int), Tm 1, Nt 0, Nt 1])]
  = [(0, [Tm 0, Nt 2]), (2, [Tm 1, Nt 3]), (3, [Nt 0, Nt 1])]
by eval
```

Firstly we show that the *binarize* function transforms a production list into a binary production list

```
lemma count_dec1:
  assumes binarize1 ps' ps ≠ ps
  shows count ps > count (binarize1 ps' ps)
using assms proof (induction ps' ps rule: binarize1.induct)
  case (3 ps' A s0 u ps)
  show ?case proof (cases length u ≤ 1)
    case True
    with 3.prems have binarize1 ps' ps ≠ ps by simp
    with True have count (binarize1 ps' ps) < count ps
```

```
    using 3.IH by simp
   with True show ?thesis by simp
  next
   case False
   let ?B = fresh0(nts ps')
   from False have count (binarize1 ps' ((A, s0 # u) # ps)) = count ((A,[s0, Nt
?B]) # (?B, u) # ps)
     by (metis binarize1.simps(3))
   also have ... = count ((?B, u) # ps) by simp
   also from False have ... < count ((A, s0 # u) # ps) by simp
   finally have count (binarize1 ps' ((A, s0 # u) # ps)) < count ((A, s0 # u)
# ps) by simp
   thus ?thesis .
  qed
qed simp_all

lemma count_dec':
  assumes binarize' ps ≠ ps
  shows count ps > count (binarize' ps)
 using binarize'_def assms count_dec1 by metis

lemma binarize_ffpi:
  binarize'((binarize' ^^ count x) x) = (binarize' ^^ count x) x
proof −
  have ⋀x. count(binarize' x) < count x ∨ binarize' x = x
    using count_dec' by blast
  thus ?thesis using funpow_fix by metis
qed

lemma binarize_binary1:
  assumes ps = binarize1 ps' ps
  shows (A,w) ∈ set(binarize1 ps' ps) ⟹ length w ≤ 2
  using assms proof (induction ps' ps rule: binarize1.induct)
  case (3 ps' C s0 u ps)
  show ?case proof (cases length u ≤ 1)
    case True
    with 3 show ?thesis by auto
  next
    case False
    hence (C, s0 # u) # ps ≠ binarize1 ps' ((C, s0 # u) # ps)
      by (metis binarize1.simps(3) list.sel(3) not_Cons_self2)
    with 3.prems(2) show ?thesis by blast
  qed
qed auto

lemma binarize_binary':
  assumes ps = binarize' ps
  shows (A,w) ∈ set(binarize' ps) ⟹ length w ≤ 2
  using binarize'_def assms binarize_binary1 by metis
```

lemma *binarize_binary*: $(A,w) \in set(binarize\ ps) \Longrightarrow length\ w \leq 2$
unfolding *binarize_def* **using** *binarize_ffpi binarize_binary'* **by** *metis*

Now we prove the property of language preservation

lemma *binarize1_cases*:
binarize1 ps' ps = ps $\vee$ $(\exists A\ ps''\ B\ u\ s.\ set\ ps = \{(A,\ s\#u)\} \cup set\ ps'' \wedge set\ (binarize1\ ps'\ ps) = \{(A,[s,Nt\ B]),(B,u)\} \cup set\ ps'' \wedge Nt\ B \notin syms\ ps')$
proof (*induction ps' ps rule*: *binarize1.induct*)
case (*2 ps' C ps*)
then show *?case*
proof (*cases binarize1 ps' ps = ps*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then obtain *A ps'' B u s* **where** *defs*: *set ps* = $\{(A,\ s\ \#\ u)\} \cup set\ ps'' \wedge set\ (binarize1\ ps'\ ps) = \{(A,\ [s,\ Nt\ B]),\ (B,\ u)\} \cup set\ ps'' \wedge Nt\ B \notin syms\ ps'$
using *2* **by** *blast*
from *defs* **have** *wit*: *set* $((C,\ []) \#\ ps) = \{(A,\ s\ \#\ u)\} \cup set\ ((C,\ []) \#\ ps'')$ **by** *simp*
from *defs* **have** *wit2*: *set* $(binarize1\ ps'\ ((C,\ []) \#\ ps)) = \{(A,\ [s,\ Nt\ B]),\ (B,\ u)\} \cup set\ ((C,\ []) \#\ ps'')$ **by** *simp*
from *defs* **have** *wit3*: *Nt B* $\notin$ *syms ps'* **by** *simp*
from *wit wit2 wit3* **show** *?thesis* **by** *blast*
qed
next
case (*3 ps' C s0 u ps*)
show *?case* **proof** (*cases length u* $\leq$ *1*)
case *T1*: *True*
then show *?thesis* **proof** (*cases binarize1 ps' ps = ps*)
case *T2*: *True*
with *T1* **show** *?thesis* **by** *simp*
next
case *False*
with *T1* **obtain** *A ps'' B v s* **where** *defs*: *set ps* = $\{(A,\ s\ \#\ v)\} \cup set\ ps'' \wedge set\ (binarize1\ ps'\ ps) = \{(A,\ [s,\ Nt\ B]),\ (B,\ v)\} \cup set\ ps'' \wedge Nt\ B \notin syms\ ps'$
using *3* **by** *blast*
from *defs* **have** *wit*: *set* $((C,\ s0\ \#\ u)\ \#\ ps) = \{(A,\ s\ \#\ v)\} \cup set\ ((C,\ s0\ \#\ u)\ \#\ ps'')$ **by** *simp*
from *defs T1* **have** *wit2*: *set* $(binarize1\ ps'\ ((C,\ s0\ \#\ u)\ \#\ ps)) = \{(A,\ [s,\ Nt\ B]),\ (B,\ v)\} \cup set\ ((C,\ s0\ \#\ u)\ \#\ ps'')$ **by** *simp*
from *defs* **have** *wit3*: *Nt B* $\notin$ *syms ps'* **by** *simp*
from *wit wit2 wit3* **show** *?thesis* **by** *blast*
qed
next
case *False*
then show *?thesis*
using *fresh0_nts in_Nts_iff_in_Syms*[*of fresh0* (*nts ps'*) *set ps'*]

by (*fastforce simp add*: *Let_def*)
qed
qed *simp*

We show that a list of terminals *map Tm x* can be derived from the original production set *ps* if and only if *map Tm x* can be derived after the transformation of the step function *binarize′*, under the assumption that the starting symbol *A* occurs in a left-hand side of at least one production in *ps*. We can then extend this property to the *binarize* function

lemma *binarize_der′*:
assumes $A \in lhss\ ps$
shows $set\ ps \vdash [Nt\ A] \Rightarrow* map\ Tm\ x \longleftrightarrow set\ (binarize'\ ps) \vdash [Nt\ A] \Rightarrow* map\ Tm\ x$
unfolding *binarize′_def* **proof** (*cases binarize1 ps ps* = *ps*)
case *False*
then obtain $C\ ps''\ B\ u\ s$ **where** *defs*: $set\ ps = \{(C, s\ \#\ u)\} \cup set\ ps'' \wedge set\ (binarize1\ ps\ ps) = \{(C, [s, Nt\ B]), (B, u)\} \cup set\ ps'' \wedge Nt\ B \notin syms\ ps$
by (*meson binarize1_cases*)
from *defs* **have** *a_not_b*: $C \neq B$ **unfolding** *Syms_def* **by** *fast*
from *defs assms* **have** *a1*: $A \neq B$ **unfolding** *Lhss_def Syms_def* **by** *auto*
from *defs* **have** *a2*: $Nt\ B \notin set\ (map\ Tm\ x)$ **by** *auto*
from *defs* **have** *a3*: $Nt\ B \notin set\ u$ **unfolding** *Syms_def* **by** *fastforce*
from *defs* **have** $set\ ps = set\ ((C, s\ \#\ u)\ \#\ ps'')$ **by** *simp*
with *defs a_not_b* **have** *a4*: $B \notin lhss\ ((C, [s, Nt\ B])\ \#\ ps'')$ **unfolding** *Lhss_def Syms_def* **by** *auto*
from *defs* **have** *notB*: $Nt\ B \notin syms\ ps''$ **by** *fastforce*
then have *1*: $set\ ps = set\ (substP\ (Nt\ B)\ u\ ((C, [s, Nt\ B])\ \#\ ps''))$ **proof** −
from *defs* **have** $set\ ps = \{(C, s\ \#\ u)\} \cup set\ ps''$ **by** *simp*
also have ... = $set\ ((C, s\#u)\ \#\ ps'')$ **by** *simp*
also have ... = $set\ ([(C, s\#u)]\ @\ ps'')$ **by** *simp*
also from *defs* **have** ... = $set\ ([(C,substs\ (Nt\ B)\ u\ [s, Nt\ B])]\ @\ ps'')$ **unfolding** *Syms_def* **by** *fastforce*
also have ... = $set\ ((substP\ (Nt\ B)\ u\ [(C, [s, Nt\ B])])\ @\ ps'')$ **by** (*simp add*: *substP_def*)
also have ... = $set\ ((substP\ (Nt\ B)\ u\ [(C, [s, Nt\ B])])\ @\ substP\ (Nt\ B)\ u\ ps'')$ **using** *notB* **by** (*simp add*: *substP_skip2*)
also have ... = $set\ (substP\ (Nt\ B)\ u\ ((C, [s, Nt\ B])\ \#\ ps''))$ **by** (*simp add*: *substP_def*)
finally show *?thesis* **.**
qed
from *defs* **have** *2*: $set\ (binarize1\ ps\ ps) = set\ ((C, [s, Nt\ B])\ \#\ (B, u)\ \#\ ps'')$ **by** *auto*
with *1 2 a1 a2 a3 a4* **show** $(set\ ps \vdash [Nt\ A] \Rightarrow* map\ Tm\ x) = (set\ (binarize1\ ps\ ps) \vdash [Nt\ A] \Rightarrow* map\ Tm\ x)$
by (*simp add*: *derives_inlining insert_commute*)
qed *simp*

lemma *lhss_binarize1*:
$lhss\ ps \subseteq lhss\ (binarize1\ ps'\ ps)$

proof (*induction ps′ ps rule*: *binarize1.induct*)
  case (*3 ps′ A s0 u ps*)
  then show *?case* **proof** (*cases length u ≤ 1*)
    case *True*
    with *3* **show** *?thesis* **by** *auto*
  next
    case *False*
    let *?B* = *fresh0*(*nts ps′*)
    have *lhss* ((*A*, *s0* # *u*) # *ps*) = {*A*} ∪ *lhss ps* **by** *simp*
    also have ... ⊆ {*A*} ∪ {*?B*} ∪ *lhss ps* **by** *blast*
    also have ... = *lhss* ((*A*,[*s0*, *Nt ?B*]) # (*?B*, *u*) # *ps*) **by** *simp*
    also from *False* **have** ... = *lhss* (*binarize1 ps′* ((*A*, *s0* # *u*) # *ps*))
      by (*metis binarize1.simps*(*3*))
    finally show *?thesis* **.**
  qed
qed *auto*

lemma *lhss_binarize′n*:
  lhss ps ⊆ *lhss* ((*binarize′*^^*n*) *ps*)
proof (*induction n*)
  case (*Suc n*)
  thus *?case* **unfolding** *binarize′_def* **using** *lhss_binarize1* **by** *auto*
qed *simp*

lemma *binarize_der′n*:
  assumes *A* ∈ *lhss ps*
  shows *set ps* ⊢ [*Nt A*] ⇒∗ *map Tm x* ⟷ *set* ((*binarize′*^^*n*) *ps*) ⊢ [*Nt A*] ⇒∗ *map Tm x*
using *assms* **proof** (*induction n*)
  case (*Suc n*)
  hence *A* ∈ *lhss* ((*binarize′* ^^ *n*) *ps*)
    using *lhss_binarize′n* **by** *blast*
  hence *set* ((*binarize′* ^^ *n*) *ps*) ⊢ [*Nt A*] ⇒∗ *map Tm x* ⟷ *set* (*binarize′* ((*binarize′* ^^ *n*) *ps*)) ⊢ [*Nt A*] ⇒∗ *map Tm x*
    using *binarize_der′* **by** *blast*
  hence *set* ((*binarize′* ^^ *n*) *ps*) ⊢ [*Nt A*] ⇒∗ *map Tm x* ⟷ *set* ((*binarize′* ^^ *Suc n*) *ps*) ⊢ [*Nt A*] ⇒∗ *map Tm x*
    by *simp*
  with *Suc* **show** *?case* **by** *blast*
qed *simp*

lemma *binarize_der*:
  assumes *A* ∈ *lhss ps*
  shows *set ps* ⊢ [*Nt A*] ⇒∗ *map Tm x* ⟷ *set* (*binarize ps*) ⊢ [*Nt A*] ⇒∗ *map Tm x*
  unfolding *binarize_def* **using** *binarize_der′n*[*OF assms*] **by** *simp*

lemma *lang_binarize_lhss*:
  assumes *A* ∈ *lhss ps*

shows *lang ps A = lang (binarize ps) A*
using *binarize_der[OF assms] Lang_eqI_derives* **by** *metis*

As a last step, we generalize the language preservation property to also include non-terminals which only occur at right-hand sides of the production set

lemma *binarize_syms1*:
assumes *Nt A ∈ syms ps*
shows *Nt A ∈ syms (binarize1 ps' ps)*
using *assms* **proof** (*induction ps' ps rule*: *binarize1.induct*)
case (*3 ps' A s0 u ps*)
show *?case* **proof** (*cases length u ≤ 1*)
case *True*
with *3* **show** *?thesis* **by** *auto*
next
case *False*
with *3* **show** *?thesis* **by** (*auto simp*: *Syms_def Let_def*)
qed
qed *auto*

lemma *binarize_lhss_nts1*:
assumes *A ∉ lhss ps*
and *A ∈ nts ps'*
shows *A ∉ lhss (binarize1 ps' ps)*
using *assms* **proof** (*induction ps' ps rule*: *binarize1.induct*)
case (*3 ps' A s0 u ps*)
thus *?case* **proof** (*cases length u ≤ 1*)
case *True*
with *3* **show** *?thesis* **by** *auto*
next
case *False*
with *3* **show** *?thesis* **by** (*auto simp add*: *Let_def fresh0_nts*)
qed
qed *simp_all*

lemma *binarize_lhss_nts'n*:
assumes *A ∉ lhss ps*
and *A ∈ nts ps*
shows *A ∉ lhss ((binarize'^^n) ps) ∧ A ∈ nts ((binarize'^^n) ps)*
using *assms* **proof** (*induction n*)
case (*Suc n*)
thus *?case*
unfolding *binarize'_def* **by** (*simp add*: *binarize_lhss_nts1 binarize_syms1 in_Nts_iff_in_Syms*)
qed *simp*

lemma *binarize_lhss_nts*:
assumes *A ∉ lhss ps*
and *A ∈ nts ps*

shows $A \notin lhss\ (binarize\ ps) \wedge A \in nts\ (binarize\ ps)$
unfolding *binarize_def* **using** *binarize_lhss_nts'n*[*OF assms*] **by** *simp*

lemma *binarize_nts'n*:
assumes $A \in nts\ ps$
shows $A \in nts\ ((binarize'\ ^\frown\ n)\ ps)$
using *assms* **proof** (*induction n*)
case (*Suc n*)
thus *?case*
unfolding *binarize'_def* **by** (*simp add*: *binarize_syms1 in_Nts_iff_in_Syms*)
qed *simp*

lemma *binarize_nts*:
assumes $A \in nts\ ps$
shows $A \in nts\ (binarize\ ps)$
unfolding *binarize_def* **using** *assms binarize_nts'n* **by** *blast*

lemma *lang_binarize*:
assumes $A \in nts\ ps$
shows $lang\ (binarize\ ps)\ A = lang\ ps\ A$
proof (*cases* $A \in lhss\ ps$)
case *True*
thus *?thesis*
using *lang_binarize_lhss* **by** *blast*
next
case *False*
thus *?thesis*
using *assms binarize_lhss_nts Lang_empty_if_notin_Lhss* **by** *fast*
qed

end

14 Right-Linear Grammars

theory *Right_Linear*
imports *Unit_Elimination Binarize*
begin

This theory defines (strongly) right-linear grammars and proves that every right-linear grammar can be transformed into a strongly right-linear grammar.

In a *right linear* grammar every production has the form $A \to wB$ or $A \to w$ where w is a sequence of terminals:

definition *rlin* :: ('*n*, '*t*) *Prods* ⇒ *bool* **where**
$rlin\ ps = (\forall (A,w) \in ps.\ \exists u.\ w = map\ Tm\ u \vee (\exists B.\ w = map\ Tm\ u\ @\ [Nt\ B]))$

definition *rlin_noterm* :: ('*n*, '*t*) *Prods* ⇒ *bool* **where**
$rlin\_noterm\ ps = (\forall (A,w) \in ps.\ w = [] \vee (\exists u\ B.\ w = map\ Tm\ u\ @\ [Nt\ B]))$

definition *rlin_bin* :: ('*n*, '*t*) *Prods* ⇒ *bool* **where**
rlin_bin ps = (∀ (*A*,*w*) ∈ *ps*. *w* = [] ∨ (∃ *B*. *w* = [*Nt B*] ∨ (∃ *a*. *w* = [*Tm a*, *Nt B*])))

In a *strongly right linear* grammar every production has the form $A \to aB$ or $A \to \varepsilon$ where a is a terminal:

definition *rlin2* :: ('*a*, '*t*) *Prods* ⇒ *bool* **where**
rlin2 ps = (∀ (*A*,*w*) ∈ *ps*. *w* = [] ∨ (∃ *a B*. *w* = [*Tm a*, *Nt B*]))

A straightforward property:

lemma *rlin_if_rlin2*:
assumes *rlin2 ps*
shows *rlin ps*
proof −
have ∃ *u*. *x2* = *map Tm u* ∨ (∃ *B*. *x2* = *map Tm u* @ [*Nt B*])
if ∀ *x*∈*ps*. ∀ *x1 x2*. *x* = (*x1*, *x2*) ⟶ *x2* = [] ∨ (∃ *a B*. *x2* = [*Tm a*, *Nt B*])
and (*x1*, *x2*) ∈ *ps*
for *x1* :: '*a* **and** *x2* :: ('*a*, '*b*) *sym list*
using *that* **by** (*metis append_Cons append_Nil list.simps*(*8*,*9*))
with *assms* **show** *?thesis*
unfolding *rlin_def rlin2_def*
by (*auto split*: *prod.splits*)
qed

lemma *rlin_cases*:
assumes *rlin_ps*: *rlin ps*
and *elem*: (*A*,*w*) ∈ *ps*
shows (∃ *B*. *w* = [*Nt B*]) ∨ (∃ *u*. *w* = *map Tm u* ∨ (∃ *B*. *w* = *map Tm u* @ [*Nt B*] ∧ *length u* > *0*))
proof −
from *rlin_ps* **have** ∀ (*A*,*w*) ∈ *ps*. (∃ *u*. *w* = *map Tm u* ∨ (∃ *B*. *w* = *map Tm u* @ [*Nt B*] ∧ *length u* ≤ *0*))
∨ (∃ *u*. *w* = *map Tm u* ∨ (∃ *B*. *w* = *map Tm u* @ [*Nt B*] ∧ *length u* > *0*))
using *rlin_def* **by** *fastforce*
with *elem* **have** (∃ *u*. *w* = *map Tm u* ∨ (∃ *B*. *w* = *map Tm u* @ [*Nt B*] ∧ *length u* ≤ *0*))
∨ (∃ *u*. *w* = *map Tm u* ∨ (∃ *B*. *w* = *map Tm u* @ [*Nt B*] ∧ *length u* > *0*)) **by** *auto*
hence (∃ *u*. *w* = *map Tm u* ∨ (∃ *B*. *w* = *map Tm u* @ [*Nt B*] ∧ *length u* = *0*))
∨ (∃ *u*. *w* = *map Tm u* ∨ (∃ *B*. *w* = *map Tm u* @ [*Nt B*] ∧ *length u* > *0*)) **by** *simp*
hence (∃ *u*. *w* = *map Tm u* ∨ (∃ *B*. *w* = [*Nt B*]))
∨ (∃ *u*. *w* = *map Tm u* ∨ (∃ *B*. *w* = *map Tm u* @ [*Nt B*] ∧ *length u* > *0*)) **by** *auto*
hence (∃ *B*. *w* = [*Nt B*]) ∨ (∃ *u*. *w* = *map Tm u* ∨ (∃ *B*. *w* = *map Tm u* @ [*Nt B*] ∧ *length u* > *0*)) **by** *blast*
thus *?thesis* **.**
qed

14.1 From *rlin* to *rlin2*

The *finalize* function is responsible of the transformation of a production list from *rlin* to *rlin_noterm*, while preserving the language. We make use of fixpoint iteration and define the function *finalize1* that adds a fresh nonterminal B at the end of every production that consists only of terminals and has at least length one. The function also introduces the new production $(B,[])$ in the production list. The step function of the fixpoint iteration is then the auxiliary function *finalize'*. We also define the count function as *countfin* which counts the the productions that consists only of terminal and has at least length one

fun *finalize1* :: ('*n* :: *fresh0*, '*t*) *prods* ⇒ ('*n*, '*t*) *prods* ⇒ ('*n*, '*t*) *prods* **where**
  finalize1 ps' [] = []
| *finalize1 ps' ((A,[])#ps) = (A,[]) # finalize1 ps' ps*
| *finalize1 ps' ((A,w)#ps) =*
  (if ∃ u. w = map Tm u then let B = fresh0(nts ps') in (A,w @ [Nt B])#(B,[])#ps else (A,w) # finalize1 ps' ps)

declare *finalize1.simps*(*1*,*2*)[*code*]

lemma *finalize1_code*[*code*]:
  finalize1 ps' ((A,x#xs) # ps) = (if nts_syms (x#xs) = {} then let B = fresh0(nts ps') in (A,(x#xs) @ [Nt B])#(B,[])#ps else (A,x#xs) # finalize1 ps' ps)
unfolding *finalize1.simps* **by**(*simp only*: *nts_syms_empty_iff*)

definition *finalize'* :: ('*n*::*fresh0*, '*t*) *prods* ⇒ ('*n*, '*t*) *prods* **where**
  finalize' ps = finalize1 ps ps

fun *countfin* :: ('*n*::*infinite*, '*t*) *prods* ⇒ *nat* **where**
  countfin [] = 0
| *countfin ((A,[])#ps) = countfin ps*
| *countfin ((A,w) # ps) = (if ∃ u. w = map Tm u then 1 + countfin ps else countfin ps)*

declare *countfin.simps*(*1*,*2*)[*code*]

lemma *countfin_code*[*code*]: *countfin ((A,x#ys) # ps) = (if nts_syms (x#ys) = {} then 1 + countfin ps else countfin ps)*
unfolding *countfin.simps* **by**(*simp only*: *nts_syms_empty_iff*)

definition *finalize* :: ('*n*::*fresh0*, '*t*) *prods* ⇒ ('*n*, '*t*) *prods* **where**
  finalize ps = (finalize' ^^ (countfin ps)) ps

Firstly we show that *finalize* indeed does the intended transformation

lemma *countfin_dec1*:
  assumes *finalize1 ps' ps ≠ ps*
  shows *countfin ps > countfin (finalize1 ps' ps)*
using *assms* **proof** (*induction ps' ps rule*: *finalize1.induct*)

```
  case (3 ps' A v va ps)
  thus ?case proof (cases ∃ u. v # va = map Tm u)
    case True
    let ?B = fresh0(nts ps')
    have not_tm: ∄ u. v # va @ [Nt ?B] = map Tm u
      by (simp add: ex_map_conv)
   from True have countfin (finalize1 ps' ((A, v # va) # ps)) = countfin ((A,v#va
@ [Nt ?B])#(?B,[])#ps)
      by (metis append_Cons finalize1.simps(3))
    also from not_tm have ... = countfin ps by simp
    also have ... < countfin ps + 1 by simp
    also from True have ... = countfin ((A, v # va) # ps) by simp
    finally show ?thesis .
  next
    case False
    with 3 show ?thesis by simp
  qed
qed simp_all

lemma countfin_dec':
  assumes finalize' ps ≠ ps
  shows countfin ps > countfin (finalize' ps)
  using finalize'_def assms countfin_dec1 by metis

lemma finalize_ffpi:
  finalize'((finalize' ^^ countfin x) x) = (finalize' ^^ countfin x) x
proof −
  have ⋀x. countfin(finalize' x) < countfin x ∨ finalize' x = x
    using countfin_dec' by blast
  thus ?thesis using funpow_fix by metis
qed

lemma finalize_rlinnoterm1:
  assumes rlin (set ps)
      and ps = finalize1 ps' ps
    shows rlin_noterm (set ps)
  using assms proof (induction ps' ps rule: finalize1.induct)
  case (1 ps')
  thus ?case
    by (simp add: rlin_noterm_def)
next
  case (2 ps' A ps)
  thus ?case
    by (simp add: rlin_def rlin_noterm_def)
next
  case (3 ps' A v va ps)
  thus ?case proof (cases ∃ u. v#va = map Tm u)
    case True
    with 3 show ?thesis
```

```
    by simp (meson list.inject not_Cons_self)
  next
    case False
    with 3 show ?thesis
      by (simp add: rlin_def rlin_noterm_def)
  qed
qed

lemma finalize_rlin1:
  rlin (set ps) ⟹ rlin (set (finalize1 ps' ps))
proof (induction ps' ps rule: finalize1.induct)
  case (2 ps' A ps)
  thus ?case
    by (simp add: rlin_def)
next
  case (3 ps' A v va ps)
  thus ?case proof (cases ∃ u. v#va = map Tm u)
    case True
    with 3 show ?thesis
    by (auto simp: Let_def rlin_def split_beta map_eq_append_conv Cons_eq_append_conv
intro: exI[of _ _ # _])
  next
    case False
    with 3 show ?thesis
      by (simp add: rlin_def)
  qed
qed simp

lemma finalize_rlin':
  rlin (set ps) ⟹ rlin (set (finalize' ps))
  unfolding finalize'_def using finalize_rlin1 by blast

lemma finalize_rlin'n:
  rlin (set ps) ⟹ rlin (set ((finalize'^^n) ps))
  by (induction n) (auto simp add: finalize_rlin')

lemma finalize_rlinnoterm':
  assumes rlin (set ps)
      and ps = finalize' ps
  shows rlin_noterm (set ps)
  using finalize'_def assms finalize_rlinnoterm1 by metis

lemma finalize_rlinnoterm:
  rlin (set ps) ⟹ rlin_noterm (set (finalize ps))
proof −
  assume asm: rlin (set ps)
  hence 1: rlin (set ((finalize' ^^ countfin ps) ps))
    using finalize_rlin'n by auto
  have finalize'((finalize' ^^ countfin ps) ps) = (finalize' ^^ countfin ps) ps
```

using *finalize_ffpi* **by** *blast*
with *1* **have** *rlin_noterm* (*set* ((*finalize'* ⁀ *countfin ps*) *ps*))
using *finalize_rlinnoterm'* **by** *metis*
hence *rlin_noterm* (*set* (*finalize ps*))
by (*simp add*: *finalize_def*)
thus *?thesis* **.**
qed

Now proving the language preservation property of *finalize*, similarly to *binarize*

lemma *finalize1_cases*:
finalize1 ps' ps = *ps* ∨ (∃ *A w ps'' B*. *set ps* = {(*A*, *w*)} ∪ *set ps''* ∧ *set* (*finalize1 ps' ps*) = {(*A*,*w* @ [*Nt B*]),(*B*,[])} ∪ *set ps''* ∧ *Nt B* ∉ *syms ps'*)
proof (*induction ps' ps rule*: *finalize1.induct*)
case (*2 ps' C ps*)
thus *?case* **proof** (*cases finalize1 ps' ps* = *ps*)
case *False*
then obtain *A w ps'' B* **where** *defs*: *set ps* = {(*A*, *w*)} ∪ *set ps''* ∧ *set* (*finalize1 ps' ps*) = {(*A*, *w* @ [*Nt B*]), (*B*, [])} ∪ *set ps''* ∧ *Nt B* ∉ *syms ps'*
using *2* **by** *blast*
from *defs* **have** *wit*: *set* ((*C*, []) # *ps*) = {(*A*, *w*)} ∪ *set* ((*C*, []) # *ps''*) **by** *simp*
from *defs* **have** *wit2*: *set* (*finalize1 ps'* ((*C*, []) # *ps*)) = {(*A*, *w* @ [*Nt B*]), (*B*, [])} ∪ *set* ((*C*, []) # *ps''*) **by** *simp*
from *defs* **have** *wit3*: *Nt B* ∉ *syms ps'* **by** *simp*
from *wit wit2 wit3* **show** *?thesis* **by** *blast*
qed *simp*
next
case (*3 ps' C v va ps*)
thus *?case* **proof** (*cases* ∃ *u*. *v*#*va* = *map Tm u*)
case *True*
thus *?thesis* **using** *fresh0_nts in_Nts_iff_in_Syms*
by (*simp add*: *Let_def*) *fastforce*
next
case *false1*: *False*
thus *?thesis* **proof** (*cases finalize1 ps' ps* = *ps*)
case *False*
with *false1* **obtain** *A w ps'' B* **where** *defs*: *set ps* = {(*A*, *w*)} ∪ *set ps''* ∧ *set* (*finalize1 ps' ps*) = {(*A*, *w* @ [*Nt B*]), (*B*, [])} ∪ *set ps''* ∧ *Nt B* ∉ *syms ps'*
using *3* **by** *blast*
from *defs* **have** *wit*: *set* ((*C*, *v*#*va*) # *ps*) = {(*A*, *w*)} ∪ *set* ((*C*, *v*#*va*) # *ps''*) **by** *simp*
from *defs false1* **have** *wit2*: *set* (*finalize1 ps'* ((*C*, *v*#*va*) # *ps*)) = {(*A*, *w* @ [*Nt B*]), (*B*, [])} ∪ *set* ((*C*, *v*#*va*) # *ps''*) **by** *simp*
from *defs* **have** *wit3*: *Nt B* ∉ *syms ps'* **by** *simp*
from *wit wit2 wit3* **show** *?thesis* **by** *blast*
qed *simp*
qed
qed *simp*

lemma *finalize_der'*:
assumes $A \in lhss\ ps$
shows $set\ ps \vdash [Nt\ A] \Rightarrow* map\ Tm\ x \longleftrightarrow set\ (finalize'\ ps) \vdash [Nt\ A] \Rightarrow* map\ Tm\ x$
unfolding *finalize'_def* **proof** (*cases finalize1 ps ps = ps*)
case *False*
then obtain $C\ w\ ps''\ B$ **where** *defs*: $set\ ps = \{(C, w)\} \cup set\ ps'' \wedge set\ (finalize1\ ps\ ps) = \{(C, w @ [Nt\ B]), (B, [])\} \cup set\ ps'' \wedge Nt\ B \notin syms\ ps$
by (*meson finalize1_cases*)
from *defs* **have** *a_not_b*: $C \neq B$ **unfolding** *Syms_def* **by** *fast*
from *defs assms* **have** *a1*: $A \neq B$ **unfolding** *Lhss_def Syms_def* **by** *auto*
from *defs* **have** *a2*: $Nt\ B \notin set\ (map\ Tm\ x)$ **by** *auto*
from *defs* **have** *a3*: $Nt\ B \notin set\ []$ **by** *simp*
from *defs* **have** $set\ ps = set\ ((C, w) \# ps'')$ **by** *simp*
with *defs a_not_b* **have** *a4*: $B \notin lhss\ ((C, w @ [Nt\ B]) \# ps'')$
unfolding *Lhss_def Syms_def* **by** *auto*
from *defs* **have** *notB*: $Nt\ B \notin syms\ ps''$ **unfolding** *Syms_def* **by** *blast*
then have *1*: $set\ ps = set\ (substP\ (Nt\ B)\ []\ ((C, w @ [Nt\ B]) \# ps''))$ **proof** −
from *defs* **have** *s1*: $Nt\ B \notin syms\ ps$ **unfolding** *Syms_def* **by** *meson*
from *defs* **have** *s2*: $(C,w) \in set\ ps$ **by** *blast*
from *s1 s2* **have** *b_notin_w*: $Nt\ B \notin set\ w$ **unfolding** *Syms_def* **by** *fastforce*
from *defs* **have** $set\ ps = \{(C, w)\} \cup set\ ps''$ **by** *simp*
also have $... = set\ ((C, w) \# ps'')$ **by** *simp*
also have $... = set\ ([(C, w)] @ ps'')$ **by** *simp*
also from *defs* **have** $... = set\ ([(C,substsNt\ B\ []\ (w @ [Nt\ B]))] @ ps'')$ **using** *b_notin_w*
by (*simp add*: *substs_skip*)
also have $... = set\ ((substP\ (Nt\ B)\ []\ [(C, w @ [Nt\ B])]) @ ps'')$ **by** (*simp add*: *substP_def*)
also have $... = set\ ((substP\ (Nt\ B)\ []\ [(C, w @ [Nt\ B])]) @ substP\ (Nt\ B)\ []\ ps'')$ **using** *notB* **by** (*simp add*: *substP_skip2*)
also have $... = set\ (substP\ (Nt\ B)\ []\ ((C, w @ [Nt\ B]) \# ps''))$ **by** (*simp add*: *substP_def*)
finally show *?thesis* **.**
qed
from *defs* **have** *2*: $set\ (finalize1\ ps\ ps) = set\ ((C, w @ [Nt\ B]) \# (B, []) \# ps'')$ **by** *auto*
with *1 2 a1 a2 a3 a4* **show** $set\ ps \vdash [Nt\ A] \Rightarrow* map\ Tm\ x \longleftrightarrow set\ (finalize1\ ps\ ps) \vdash [Nt\ A] \Rightarrow* map\ Tm\ x$
by (*simp add*: *derives_inlining insert_commute*)
qed *simp*

lemma *lhss_finalize1*:
$lhss\ ps \subseteq lhss\ (finalize1\ ps'\ ps)$
proof (*induction ps' ps rule*: *finalize1.induct*)
case (*2 ps' A ps*)
thus *?case* **unfolding** *Lhss_def* **by** *auto*
next

case (*3 ps′ A v va ps*)
thus *?case* **unfolding** *Lhss_def* **by** (*auto simp*: *Let_def*)
qed *simp*

lemma *lhss_binarize′n*:
lhss ps ⊆ *lhss* ((*finalize′*^^*n*) *ps*)
proof (*induction n*)
case (*Suc n*)
thus *?case* **unfolding** *finalize′_def* **using** *lhss_finalize1* **by** *auto*
qed *simp*

lemma *finalize_der′n*:
assumes *A* ∈ *lhss ps*
shows *set ps* ⊢ [*Nt A*] ⇒∗ *map Tm x* ⟷ *set* ((*finalize′*^^*n*) *ps*) ⊢ [*Nt A*] ⇒∗ *map Tm x*
using *assms* **proof** (*induction n*)
case (*Suc n*)
hence *A* ∈ *lhss* ((*finalize′* ^^ *n*) *ps*)
using *lhss_binarize′n* **by** *blast*
hence *set* ((*finalize′* ^^ *n*) *ps*) ⊢ [*Nt A*] ⇒∗ *map Tm x* ⟷ *set* (*finalize′* ((*finalize′* ^^ *n*) *ps*)) ⊢ [*Nt A*] ⇒∗ *map Tm x*
using *finalize_der′* **by** *blast*
hence*set* ((*finalize′* ^^ *n*) *ps*) ⊢ [*Nt A*] ⇒∗ *map Tm x* ⟷ *set* ((*finalize′* ^^ *Suc n*) *ps*) ⊢ [*Nt A*] ⇒∗ *map Tm x*
by *simp*
with *Suc* **show** *?case* **by** *blast*
qed *simp*

lemma *finalize_der*:
assumes *A* ∈ *lhss ps*
shows *set ps* ⊢ [*Nt A*] ⇒∗ *map Tm x* ⟷ *set* (*finalize ps*) ⊢ [*Nt A*] ⇒∗ *map Tm x*
unfolding *finalize_def* **using** *finalize_der′n*[*OF assms*] **by** *simp*

lemma *lang_finalize_lhss*:
assumes *A* ∈ *lhss ps*
shows *lang ps A* = *lang* (*finalize ps*) *A*
using *finalize_der*[*OF assms*] *Lang_eqI_derives* **by** *metis*

lemma *finalize_syms1*:
assumes *Nt A* ∈ *syms ps*
shows *Nt A* ∈ *syms* (*finalize1 ps′ ps*)
using *assms* **proof** (*induction ps′ ps rule*: *finalize1.induct*)
case (*3 ps′ A v va ps*)
thus *?case* **proof** (*cases* ∃ *u*. *v*#*va* = *map Tm u*)
case *True*
with *3* **show** *?thesis* **unfolding** *Syms_def* **by** (*auto simp*: *Let_def*)
next
case *False*

```
    with 3 show ?thesis unfolding Syms_def by auto
  qed
qed auto

lemma finalize_nts'n:
  assumes A ∈ nts ps
  shows   A ∈ nts ((finalize' ^^ n) ps)
  using assms proof (induction n)
  case (Suc n)
  thus ?case
    unfolding finalize'_def by (simp add: finalize_syms1 in_Nts_iff_in_Syms)
qed simp

lemma finalize_nts:
  assumes A ∈ nts ps
  shows   A ∈ nts (finalize ps)
  unfolding finalize_def using finalize_nts'n[OF assms] by simp

lemma finalize_lhss_nts1:
  assumes A ∉ lhss ps
      and A ∈ nts ps'
    shows A ∉ lhss (finalize1 ps' ps)
using assms proof (induction ps' ps rule: finalize1.induct)
  case (3 ps' A v va ps)
  thus ?case proof (cases ∃ u. v#va = map Tm u)
    case True
   with 3 show ?thesis unfolding Lhss_def by (auto simp: Let_def fresh0_nts)
  next
    case False
    with 3 show ?thesis unfolding Lhss_def by (auto simp: Let_def)
  qed
qed simp_all

lemma finalize_lhss_nts'n:
  assumes A ∉ lhss ps
      and A ∈ nts ps
    shows A ∉ lhss ((finalize'^^n) ps) ∧ A ∈ nts ((finalize'^^n) ps)
  using assms proof (induction n)
  case (Suc n)
  thus ?case
   unfolding finalize'_def by (simp add: finalize_lhss_nts1 finalize_syms1 in_Nts_iff_in_Syms)
qed simp

lemma finalize_lhss_nts:
   assumes A ∉ lhss ps
      and  A ∈ nts ps
    shows A ∉ lhss (finalize ps) ∧ A ∈ nts (finalize ps)
  unfolding finalize_def using finalize_lhss_nts'n[OF assms] by simp
```

lemma *lang_finalize*:
assumes $A \in nts\ ps$
shows $lang\ (finalize\ ps)\ A = lang\ ps\ A$
proof (*cases* $A \in lhss\ ps$)
case *True*
thus *?thesis*
using *lang_finalize_lhss* **by** *blast*
next
case *False*
thus *?thesis*
using *assms finalize_lhss_nts Lang_empty_if_notin_Lhss* **by** *fast*
qed

Next step is to define the transformation from *rlin_noterm* to *rlin_bin*. For this we use the function *binarize*. The language preservation property of *binarize* is already proven

lemma *binarize_rlinbin1*:
assumes *rlin_noterm* (*set ps*)
and $ps = binarize1\ ps'\ ps$
shows *rlin_bin* (*set ps*)
using *assms* **proof** (*induction ps' ps rule*: *binarize1.induct*)
case (*1 ps'*)
thus *?case*
by (*simp add*: *rlin_bin_def*)
next
case (*2 ps' A ps*)
thus *?case*
by (*simp add*: *rlin_noterm_def rlin_bin_def*)
next
case (*3 ps' A s0 u ps*)
from *3.prems*(*2*) **have** *a1*: $length\ u \leq 1$ **by** *simp* (*meson list.inject not_Cons_self*)
with *3.prems*(*2*) **have** *a2*: $ps = binarize1\ ps'\ ps$ **by** *simp*
from *3.prems*(*1*) **have** *a3*: *rlin_noterm* (*set ps*)
by (*simp add*: *rlin_noterm_def*)
from *a1 a2 a3* **have** *1*: *rlin_bin* (*set ps*)
using *3.IH* **by** *blast*
from *3.prems*(*1*) **have** *ex*: $\exists\, v\ B.\ s0\ \#\ u = map\ Tm\ v\ @\ [Nt\ B]$
by (*simp add*: *rlin_noterm_def*)
with *a1* **have** *2*: $\exists\, B.\ s0\ \#\ u = [Nt\ B] \vee (\exists\, a.\ s0\ \#\ u = [Tm\ a,\ Nt\ B])$ **proof**
(*cases* $length\ u = 0$)
case *True*
with *ex* **show** *?thesis* **by** *simp*
next
case *False*
with *a1* **have** $length\ u = 1$ **by** *linarith*
show *?thesis*
proof −
have $\exists\, B.\ s0 = Nt\ B \wedge u = [] \vee (\exists\, a.\ s0 = Tm\ a) \wedge u = [Nt\ B]$
if $length\ u = Suc\ 0$ **and** $s0\ \#\ u = map\ Tm\ v\ @\ [Nt\ B]$

```
      for v :: 'b list and B :: 'a
     using that by (metis append_Cons append_Nil append_butlast_last_id but-
last_snoc diff_Suc_1 hd_map last_snoc length_0_conv length_butlast list.sel(1)
list.simps(8))
    with ex ‹length u = 1› show ?thesis
     by auto
   qed
  qed
  from 1 2 show ?case
   by (simp add: rlin_bin_def)
qed

lemma binarize_noterm1:
  rlin_noterm (set ps) ⟹ rlin_noterm (set (binarize1 ps' ps))
proof (induction ps' ps rule: binarize1.induct)
  case (2 ps' A ps)
  thus ?case
   by (simp add: rlin_noterm_def)
next
  case (3 ps' A s0 u ps)
  thus ?case proof (cases length u ≤ 1)
   case True
   with 3 show ?thesis
    by (simp add: rlin_noterm_def)
  next
   case False
   let ?B = fresh0(nts ps')
   from 3.prems have a1: rlin_noterm (set ps)
    by (simp add: rlin_noterm_def)
   from 3.prems have ex: ∃ v B. s0 # u = map Tm v @ [Nt B]
    by (simp add: rlin_noterm_def)
   with False have a2: ∃ v B. [s0, Nt ?B] = map Tm v @ [Nt B]
    by (auto simp: Cons_eq_append_conv neq_Nil_conv intro: exI[of _ [_]])
   from ex False have a3: ∃ v B. u = map Tm v @ [Nt B]
    by (auto simp: Cons_eq_append_conv)
   from False a1 a2 a3 show ?thesis
    by (auto simp: Let_def rlin_noterm_def)
  qed
qed simp

lemma binarize_noterm':
  rlin_noterm (set ps) ⟹ rlin_noterm (set (binarize' ps))
  unfolding binarize'_def using binarize_noterm1 by blast

lemma binarize_noterm'n:
  rlin_noterm (set ps) ⟹ rlin_noterm (set ((binarize'^^n) ps))
  by (induction n) (auto simp add: binarize_noterm')

lemma binarize_rlinbin':
```

assumes *rlin_noterm* (*set ps*)
and *ps* = *binarize′ ps*
shows *rlin_bin* (*set ps*)
using *binarize′_def assms binarize_rlinbin1* **by** *metis*

lemma *binarize_rlinbin*:
rlin_noterm (*set ps*) ⟹ *rlin_bin* (*set* (*binarize ps*))
proof −
assume *asm*: *rlin_noterm* (*set ps*)
hence *1*: *rlin_noterm* (*set* ((*binarize′* ^^ *count ps*) *ps*))
using *binarize_noterm′n* **by** *auto*
have *binarize′*((*binarize′* ^^ *count ps*) *ps*) = (*binarize′* ^^ *count ps*) *ps*
using *binarize_ffpi* **by** *blast*
with *1* **have** *rlin_bin* (*set* ((*binarize′* ^^ *count ps*) *ps*))
using *binarize_rlinbin′* **by** *fastforce*
hence *rlin_bin* (*set* (*binarize ps*))
by (*simp add*: *binarize_def*)
thus *?thesis* **.**
qed

The last transformation takes a production set from *rlin_bin* and converts it to *rlin2*. That is, we need to remove unit productions of the from (*A*, [*Nt B*]). In *uProds.thy* is the predicate $\mathcal{U}$ *ps′ ps* defined that is satisfied if *ps* is the same production set as *ps′* without the unit productions. The language preservation property is already given

lemma *uppr_rlin2*:
assumes *rlinbin*: *rlin_bin* (*set ps′*)
and *uppr_ps′*: *unit_elim_rel ps′ ps*
shows *rlin2* (*set ps*)
proof −
from *rlinbin* **have** *rlin2* (*set ps′* − {(*A*,*w*) ∈ *set ps′*. ∃ *B*. *w* = [*Nt B*]})
using *rlin2_def rlin_bin_def* **by** *fastforce*
hence *rlin2* (*set ps′* − (*unit_prods ps′*))
by (*simp add*: *unit_prods_def*)
hence *1*: *rlin2* (*unit_rm ps′*)
by (*simp add*: *unit_rm_def*)
hence *2*: *rlin2* (*new_prods ps′*)
unfolding *new_prods_def rlin2_def* **by** *fastforce*
from *1 2* **have** *rlin2* (*unit_rm ps′* ∪ *new_prods ps′*)
unfolding *rlin2_def* **by** *auto*
with *uppr_ps′* **have** *rlin2* (*set ps*)
by (*simp add*: *unit_elim_rel_def*)
thus *?thesis* **.**
qed

The transformation *rlin2_of_rlin* applies the presented functions in the right order. At the end, we show that *rlin2_of_rlin* converts a production set from *rlin* to a production set from *rlin2*, without changing the language

definition *rlin2_of_rlin* :: (*′n*::*fresh0*,*′t*) *prods* ⇒ (*′n*,*′t*)*prods* **where**

rlin2_of_rlin ps = *unit_elim* (*binarize* (*finalize ps*))

lemma *binarize*(*finalize* [(*0*::*nat*, [*Tm* (*0*::*int*), *Tm 1*, *Tm 2*])])
= [(*0*, [*Tm 0*, *Nt 2*]), (*2*, [*Tm 1*, *Nt 3*]), (*3*, [*Tm 2*, *Nt 1*]), (*1*, [])]
by *eval*

theorem *rlin_to_rlin2*:
assumes *rlin* (*set ps*)
shows *rlin2* (*set* (*rlin2_of_rlin ps*))
using *assms* **proof** −
assume *rlin* (*set ps*)
hence *rlin_noterm* (*set* (*finalize ps*))
using *finalize_rlinnoterm* **by** *blast*
hence *rlin_bin* (*set* (*binarize* (*finalize ps*)))
by (*simp add*: *binarize_rlinbin*)
hence *rlin2* (*set* (*unit_elim* (*binarize* (*finalize ps*))))
by (*simp add*: *unit_elim_rel_unit_elim uppr_rlin2*)
thus *rlin2* (*set* (*rlin2_of_rlin ps*))
by (*simp add*: *rlin2_of_rlin_def*)
qed

lemma *lang_rlin2_of_rlin*:
A ∈ *Nts* (*set ps*) ⟹ *lang* (*rlin2_of_rlin ps*) *A* = *lang ps A*
by(*simp add*: *rlin2_of_rlin_def lang_unit_elim finalize_nts lang_binarize lang_finalize*)

14.2 Properties of *rlin2* derivations

In the following we present some properties for list of symbols that are derived from a production set satisfying *rlin2*

lemma *map_Tm_single_nt*:
assumes *map Tm w* @ [*Tm a*, *Nt A*] = *u1* @ [*Nt B*] @ *u2*
shows *u1* = *map Tm* (*w* @ [*a*]) ∧ *u2* = []
proof −
from *assms* **have** ∗: *map Tm* (*w* @ [*a*]) @ [*Nt A*] = *u1* @ [*Nt B*] @ *u2* **by** *simp*
have *1*: *Nt B* ∉ *set* (*map Tm* (*w* @ [*a*])) **by** *auto*
have *2*: *Nt B* ∈ *set* (*u1* @ [*Nt B*] @ *u2*) **by** *simp*
from ∗ *1 2* **have** *Nt B* ∈ *set* ([*Nt A*])
by (*metis list.set_intros*(*1*) *rotate1.simps*(*2*) *set_ConsD set_rotate1 sym.inject*(*1*))
hence [*Nt B*] = [*Nt A*] **by** *simp*
with *1* ∗ **show** *?thesis*
by (*metis append_Cons append_Cons_eq_iff append_self_conv emptyE empty_set*)
qed

A non-terminal can only occur as the rightmost symbol

lemma *rlin2_derive*:
assumes *P* ⊢ *v1* ⇒∗ *v2*
and *v1* = [*Nt A*]
and *v2* = *u1* @ [*Nt B*] @ *u2*

```
    and rlin2 P
  shows ∃ w. u1 = map Tm w ∧ u2 = []
using assms proof (induction arbitrary: u1 B u2 rule: derives_induct)
 case base
 then show ?case
  by (simp add: append_eq_Cons_conv)
next
 case (step u C v w)
 from step.prems(1) step.prems(3) have ∃ w. u = map Tm w ∧ v = []
  using step.IH[of u C v] by simp
 then obtain wh where u_def: u = map Tm wh by blast
 have v_eps: v = []
  using ‹∃ w. u = map Tm w ∧ v = []› by simp
 from step.hyps(2) step.prems(3) have w_cases: w = [] ∨ (∃ d D. w = [Tm d,
Nt D])
  unfolding rlin2_def by auto
 then show ?case proof cases
  assume w=[]
  with v_eps step.prems(2) have u = u1 @ [Nt B] @ u2 by simp
  with u_def show ?thesis by (auto simp: append_eq_map_conv)
 next
  assume ¬w=[]
  then obtain d D where w = [Tm d, Nt D]
   using w_cases by blast
  with u_def v_eps step.prems(2) have u1 = map Tm (wh @ [d]) ∧ u2 = []
   using map_Tm_single_nt[of wh d D u1 B u2] by simp
  thus ?thesis by blast
 qed
qed
```

A new terminal is introduced by a production of the form $(C, [Tm\ x, Nt\ B])$

```
lemma rlin2_introduce_tm:
 assumes rlin2 P
     and P ⊢ [Nt A] ⇒* map Tm w @ [Tm x, Nt B]
   shows ∃ C. P ⊢ [Nt A] ⇒* map Tm w @ [Nt C] ∧ (C,[Tm x, Nt B]) ∈ P
proof −
 from assms(2) have ∃ v. P ⊢ [Nt A] ⇒* v ∧ P ⊢ v ⇒ map Tm w @ [Tm x, Nt
B]
  using rtranclp.cases by fastforce
 then obtain v where v_star: P ⊢ [Nt A] ⇒* v and v_step: P ⊢ v ⇒ map Tm
w @ [Tm x, Nt B] by blast
 from v_step have ∃ u1 u2 C α. v = u1 @ [Nt C] @ u2 ∧ map Tm w @ [Tm x,
Nt B] = u1 @ α @ u2 ∧ (C,α) ∈ P
  using derive.cases by fastforce
 then obtain u1 u2 C α where v_def: v = u1 @ [Nt C] @ u2 and w_def: map
Tm w @ [Tm x, Nt B] = u1 @ α @ u2
                    and C_prod: (C,α) ∈ P by blast
 from assms(1) v_star v_def have u2_eps: u2 = []
```

```
    using rlin2_derive[of P [Nt A]] by simp
  from assms(1) v_star v_def obtain wa where u1_def: u1 = map Tm wa
    using rlin2_derive[of P [Nt A] u1 @ [Nt C] @ u2 A u1] by auto
  from w_def u2_eps u1_def have map Tm w @ [Tm x, Nt B] = map Tm wa @
α by simp
  then have map Tm (w @ [x]) @ [Nt B] = map Tm wa @ α by simp
  then have α ≠ []
   by (metis append.assoc append.right_neutral list.distinct(1) map_Tm_single_nt)
  with assms(1) C_prod obtain d D where α = [Tm d, Nt D]
    using rlin2_def by fastforce
  from w_def u2_eps have x_d: x = d
    using ‹α = [Tm d, Nt D]› by simp
  from w_def u2_eps have B_D: B = D
    using ‹α = [Tm d, Nt D]› by simp
  from x_d B_D have alpha_def: α = [Tm x, Nt B]
    using ‹α = [Tm d, Nt D]› by simp
  from w_def u2_eps alpha_def have map Tm w = u1 by simp
  with u1_def have w_eq_wa: w = wa
    by (metis list.inj_map_strong sym.inject(2))
  from v_def u1_def w_eq_wa u2_eps have v = map Tm w @ [Nt C] by simp
  with v_star have 1: P ⊢ [Nt A] ⇒* map Tm w @ [Nt C] by simp
  from C_prod alpha_def have 2: (C,[Tm x, Nt B]) ∈ P by simp
  from 1 2 show ?thesis by auto
qed

lemma rlin2_nts_derive_eq:
  assumes rlin2 P
      and P ⊢ [Nt A] ⇒* [Nt B]
    shows A = B
proof −
  from assms(2) have star_cases: [Nt A] = [Nt B] ∨ (∃ w. P ⊢ [Nt A] ⇒ w ∧ P
⊢ w ⇒* [Nt B])
    using converse_rtranclpE by force
  show ?thesis proof cases
    assume ¬[Nt A] = [Nt B]
    then obtain w where w_step: P ⊢ [Nt A] ⇒ w and w_star: P ⊢ w ⇒* [Nt
B]
      using star_cases by auto
    from assms(1) w_step have w_cases: w = [] ∨ (∃ a C. w = [Tm a, Nt C])
      unfolding rlin2_def using derive_singleton[of P Nt A w] by auto
    show ?thesis proof cases
      assume w = []
      with w_star show ?thesis by simp
    next
      assume ¬w = []
      with w_cases obtain a C where w = [Tm a, Nt C] by blast
      with w_star show ?thesis
        using derives_Tm_Cons[of P a [Nt C] [Nt B]] by simp
    qed
```

qed *simp*
qed

If the list of symbols consists only of terminals, the last production used is of the form B,[]

lemma *rlin2_tms_eps*:
assumes *rlin2 P*
and $P \vdash [Nt\ A] \Rightarrow* map\ Tm\ w$
shows $\exists B.\ P \vdash [Nt\ A] \Rightarrow* map\ Tm\ w\ @\ [Nt\ B] \wedge (B,[]) \in P$
proof −
from *assms(2)* **have** $\exists v.\ P \vdash [Nt\ A] \Rightarrow* v \wedge P \vdash v \Rightarrow map\ Tm\ w$
using *rtranclp.cases* **by** *force*
then obtain *v* **where** *v_star*: $P \vdash [Nt\ A] \Rightarrow* v$ **and** *v_step*: $P \vdash v \Rightarrow map\ Tm\ w$ **by** *blast*
from *v_step* **have** $\exists u1\ u2\ C\ \alpha.\ v = u1\ @\ [Nt\ C]\ @\ u2 \wedge map\ Tm\ w = u1\ @\ \alpha\ @\ u2 \wedge (C,\alpha) \in P$
using *derive.cases* **by** *fastforce*
then obtain *u1 u2 C* α **where** *v_def*: $v = u1\ @\ [Nt\ C]\ @\ u2$ **and** *w_def*: $map\ Tm\ w = u1\ @\ \alpha\ @\ u2$ **and** *C_prod*: $(C,\alpha) \in P$ **by** *blast*
have $\nexists A.\ Nt\ A \in set\ (map\ Tm\ w)$ **by** *auto*
with *w_def* **have** $\nexists A.\ Nt\ A \in set\ \alpha$
by (*metis Un_iff set_append*)
then have $\nexists a\ A.\ \alpha = [Tm\ a,\ Nt\ A]$ **by** *auto*
with *assms(1) C_prod* **have** *alpha_eps*: $\alpha = []$
using *rlin2_def* **by** *force*
from *v_star assms(1) v_def* **have** *u2_eps*: $u2 = []$
using *rlin2_derive*[*of P* [*Nt A*]] **by** *simp*
from *w_def alpha_eps u2_eps* **have** *u1_def*: $u1 = map\ Tm\ w$ **by** *simp*
from *v_star v_def u1_def u2_eps* **have** *1*: $P \vdash [Nt\ A] \Rightarrow* map\ Tm\ w\ @\ [Nt\ C]$ **by** *simp*
from *alpha_eps C_prod* **have** *2*: $(C,[]) \in P$ **by** *simp*
from *1 2* **show** *?thesis* **by** *auto*
qed

end

15 Strongly Right-Linear Grammars as a Nondeterministic Automaton

theory *NDA_rlin2*
imports *Right_Linear*
begin

We define what is essentially the extended transition function of a nondeterministic automaton but is driven by a set of strongly right-linear productions P, which are of course just another representation of the transitions of a nondeterministic automaton. Function *nxts_rlin2_set P M w* traverses the terminals list w starting from the set of non-terminals M according to

the productions of P. At the end it returns the reachable non-terminals.

definition *nxt_rlin2* :: $('n,'t)Prods \Rightarrow 'n \Rightarrow 't \Rightarrow 'n\ set$ **where**
nxt_rlin2 P A a = $\{B.\ (A, [Tm\ a, Nt\ B]) \in P\}$

definition *nxt_rlin2_set* :: $('n,'t)Prods \Rightarrow 'n\ set \Rightarrow 't \Rightarrow 'n\ set$ **where**
nxt_rlin2_set P M a = $(\bigcup A{\in}M.\ nxt\_rlin2\ P\ A\ a)$

definition *nxts_rlin2_set* :: $('n,'t)Prods \Rightarrow 'n\ set \Rightarrow 't\ list \Rightarrow 'n\ set$ **where**
nxts_rlin2_set P = *foldl* (*nxt_rlin2_set P*)

lemma *nxt_rlin2_nts*:
assumes $B{\in}nxt\_rlin2\ P\ A\ a$
shows $B{\in}Nts\ P$
using *assms nxt_rlin2_def Nts_def nts_syms_def* **by** *fastforce*

lemma *nxts_rlin2_set_app*:
nxts_rlin2_set P M (*x* @ *y*) = *nxts_rlin2_set P* (*nxts_rlin2_set P M x*) *y*
unfolding *nxts_rlin2_set_def* **by** *simp*

lemma *nxt_rlin2_set_pick*:
assumes $B \in nxt\_rlin2\_set\ P\ M\ a$
shows $\exists\, C{\in}M.\ B \in nxt\_rlin2\_set\ P\ \{C\}\ a$
using *assms* **by** (*simp add*:*nxt_rlin2_def nxt_rlin2_set_def*)

lemma *nxts_rlin2_set_pick*:
assumes $B \in nxts\_rlin2\_set\ P\ M\ w$
shows $\exists\, C{\in}M.\ B \in nxts\_rlin2\_set\ P\ \{C\}\ w$
using *assms* **proof** (*induction w arbitrary*: *B rule*: *rev_induct*)
case *Nil*
then show *?case*
by (*simp add*: *nxts_rlin2_set_def*)
next
case (*snoc x xs*)
from *snoc*(*2*) **have** *B_in*: $B \in nxts\_rlin2\_set\ P\ (nxts\_rlin2\_set\ P\ M\ xs)\ [x]$
using *nxts_rlin2_set_app*[*of P M xs* [*x*]] **by** *simp*
hence $B \in nxt\_rlin2\_set\ P\ (nxts\_rlin2\_set\ P\ M\ xs)\ x$
by (*simp add*: *nxts_rlin2_set_def*)
hence $\exists\, C{\in}(nxts\_rlin2\_set\ P\ M\ xs).\ B \in nxt\_rlin2\_set\ P\ \{C\}\ x$
using *nxt_rlin2_set_pick*[*of B P nxts_rlin2_set P M xs x*] **by** *simp*
then obtain *C* **where** *C_def*: $C \in nxts\_rlin2\_set\ P\ M\ xs$ **and** *C_path*: $B \in nxt\_rlin2\_set\ P\ \{C\}\ x$
by *blast*
have $\exists\, Ca{\in}M.\ C \in nxts\_rlin2\_set\ P\ \{Ca\}\ xs$
using *snoc.IH*[*of C*, *OF C_def*] **.**
then obtain *D* **where** $*$: $D \in M$ **and** *D_path*: $C \in nxts\_rlin2\_set\ P\ \{D\}\ xs$
by *blast*
from *C_path D_path* **have** $**$: $B \in nxts\_rlin2\_set\ P\ \{D\}\ (xs\ @\ [x])$
unfolding *nxts_rlin2_set_def nxt_rlin2_set_def* **by** *auto*
from $*$ $**$ **show** *?case* **by** *blast*

qed

lemma *nxts_rlin2_set_first_step*:
assumes $B \in nxts\_rlin2\_set\ P\ \{A\}\ (a\ \#\ w)$
shows $\exists\, C \in nxt\_rlin2\ P\ A\ a.\ B \in nxts\_rlin2\_set\ P\ \{C\}\ w$
proof −
from *assms* **have** $B \in nxts\_rlin2\_set\ P\ \{A\}\ ([a]@w)$ **by** *simp*
hence $B \in nxts\_rlin2\_set\ P\ (nxts\_rlin2\_set\ P\ \{A\}\ [a])\ w$
using *nxts_rlin2_set_app*[*of* $P\ \{A\}\ [a]\ w$] **by** *simp*
hence $B \in nxts\_rlin2\_set\ P\ (nxt\_rlin2\ P\ A\ a)\ w$
by (*simp add*: *nxt_rlin2_set_def nxts_rlin2_set_def*)
thus $\exists\, C \in nxt\_rlin2\ P\ A\ a.\ B \in nxts\_rlin2\_set\ P\ \{C\}\ w$
using *nxts_rlin2_set_pick*[*of* $B\ P\ nxt\_rlin2\ P\ A\ a\ w$] **by** *simp*
qed

lemma *nxts_trans0*:
assumes $B \in nxts\_rlin2\_set\ P\ (nxts\_rlin2\_set\ P\ \{A\}\ x)\ z$
shows $B \in nxts\_rlin2\_set\ P\ \{A\}\ (x@z)$
by (*metis assms foldl_append nxts_rlin2_set_def*)

lemma *nxt_mono*:
assumes $A \subseteq B$
shows $nxt\_rlin2\_set\ P\ A\ a \subseteq nxt\_rlin2\_set\ P\ B\ a$
unfolding *nxt_rlin2_set_def* **using** *assms* **by** *blast*

lemma *nxts_mono*:
assumes $A \subseteq B$
shows $nxts\_rlin2\_set\ P\ A\ w \subseteq nxts\_rlin2\_set\ P\ B\ w$
unfolding *nxts_rlin2_set_def* **proof** (*induction w rule*:*rev_induct*)
case *Nil*
thus *?case* **by** (*simp add*: *assms*)
next
case (*snoc x xs*)
thus *?case*
using *nxt_mono*[*of foldl* (*nxt_rlin2_set P*) *A xs foldl* (*nxt_rlin2_set P*) *B xs P x*] **by** *simp*
qed

lemma *nxts_trans1*:
assumes $M \subseteq nxts\_rlin2\_set\ P\ \{A\}\ x$
and $B \in nxts\_rlin2\_set\ P\ M\ z$
shows $B \in nxts\_rlin2\_set\ P\ \{A\}\ (x@z)$
using *assms nxts_trans0*[*of B P A x z*] *nxts_mono*[*of M nxts_rlin2_set P* $\{A\}$ *x P z*, *OF assms*(*1*)] **by** *auto*

lemma *nxts_trans2*:
assumes $C \in nxts\_rlin2\_set\ P\ \{A\}\ x$
and $B \in nxts\_rlin2\_set\ P\ \{C\}\ z$
shows $B \in nxts\_rlin2\_set\ P\ \{A\}\ (x@z)$

using *assms nxts_trans1*[*of* {*C*} *P A x B z*] **by** *auto*

lemma *nxts_to_mult_derive*:
B ∈ *nxts_rlin2_set P M w* ⟹ (∃ *A*∈*M*. *P* ⊢ [*Nt A*] ⇒∗ *map Tm w* @ [*Nt B*])
unfolding *nxts_rlin2_set_def* **proof** (*induction w arbitrary*: *B rule*: *rev_induct*)
case *Nil*
hence *1*: *B* ∈ *M* **by** *simp*
have *2*: *P* ⊢ [*Nt B*] ⇒∗ *map Tm* [] @ [*Nt B*] **by** *simp*
from *1 2* **show** *?case* **by** *blast*
next
case (*snoc x xs*)
from *snoc.prems* **have** ∃ *C*. *C* ∈ *foldl* (*nxt_rlin2_set P*) *M xs* ∧ (*C*, [*Tm x*, *Nt B*]) ∈ *P*
unfolding *nxt_rlin2_set_def nxt_rlin2_def* **by** *auto*
then obtain *C* **where** *C_xs*: *C* ∈ *foldl* (*nxt_rlin2_set P*) *M xs* **and** *C_prod*: (*C*, [*Tm x*, *Nt B*]) ∈ *P* **by** *blast*
from *C_xs* **obtain** *A* **where** *A_der*: *P* ⊢ [*Nt A*] ⇒∗ *map Tm xs* @ [*Nt C*] **and** *A_in*: *A* ∈ *M*
using *snoc.IH*[*of C*] **by** *auto*
from *C_prod* **have** *P* ⊢ [*Nt C*] ⇒ [*Tm x*, *Nt B*]
using *derive_singleton*[*of P Nt C* [*Tm x*, *Nt B*]] **by** *blast*
hence *P* ⊢ *map Tm xs* @ [*Nt C*] ⇒ *map Tm xs* @ [*Tm x*, *Nt B*]
using *derive_prepend*[*of P* [*Nt C*] [*Tm x*, *Nt B*] *map Tm xs*] **by** *simp*
hence *C_der*: *P* ⊢ *map Tm xs* @ [*Nt C*] ⇒ *map Tm* (*xs* @ [*x*]) @ [*Nt B*] **by** *simp*
from *A_der C_der* **have** *P* ⊢ [*Nt A*] ⇒∗ *map Tm* (*xs* @ [*x*]) @ [*Nt B*] **by** *simp*
with *A_in* **show** *?case* **by** *blast*
qed

lemma *mult_derive_to_nxts*:
assumes *rlin2 P*
shows *A*∈*M* ⟹ *P* ⊢ [*Nt A*] ⇒∗ *map Tm w* @ [*Nt B*] ⟹ *B* ∈ *nxts_rlin2_set P M w*
unfolding *nxts_rlin2_set_def* **proof** (*induction w arbitrary*: *B rule*: *rev_induct*)
case *Nil*
with *assms* **have** *A* = *B*
using *rlin2_nts_derive_eq*[*of P A B*] **by** *simp*
with *Nil.prems*(*1*) **show** *?case* **by** *simp*
next
case (*snoc x xs*)
from *snoc.prems*(*2*) **have** *P* ⊢ [*Nt A*] ⇒∗ *map Tm xs* @ [*Tm x*,*Nt B*] **by** *simp*
with *assms* **obtain** *C* **where** *C_der*: *P* ⊢ [*Nt A*] ⇒∗ *map Tm xs* @ [*Nt C*]
and *C_prods*: (*C*,[*Tm x*, *Nt B*]) ∈ *P* **using** *rlin2_introduce_tm*[*of P A xs x B*] **by** *fast*
from ‹*A* ∈ *M*› *C_der* **have** *C* ∈ *foldl* (*nxt_rlin2_set P*) *M xs*
using *snoc.IH*[*of C*] **by** *auto*
with *C_prods* **show** *?case*
unfolding *nxt_rlin2_set_def nxt_rlin2_def* **by** *auto*
qed

Acceptance of a word w w.r.t. P (starting from A), *accepted P A w*, means that we can reach an "accepting" nonterminal Z, i.e. one with a production $(Z,[])$. On the automaton level Z reachable final state. We show that *accepted P A w* iff w is in the language of A w.r.t. P.

definition *accepted P A w* = $(\exists\, Z \in$ *nxts_rlin2_set P* $\{A\}$ *w*. $(Z,[]) \in P)$

theorem *accepted_if_Lang*:
assumes *rlin2 P*
and $w \in$ *Lang P A*
shows *accepted P A w*
proof −
from *assms* **obtain** *B* **where** *A_der*: $P \vdash [Nt\ A] \Rightarrow* map\ Tm\ w\ @\ [Nt\ B]$ **and** *B_in*: $(B,[]) \in P$
unfolding *Lang_def* **using** *rlin2_tms_eps*[*of P A w*] **by** *auto*
from *A_der* **have** $B \in$ *nxts_rlin2_set P* $\{A\}$ *w*
using *mult_derive_to_nxts*[*OF assms*(*1*)] **by** *auto*
with *B_in* **show** *?thesis*
unfolding *accepted_def* **by** *blast*
qed

theorem *Lang_if_accepted*:
assumes *accepted P A w*
shows $w \in$ *Lang P A*
proof −
from *assms* **obtain** *Z* **where** *Z_nxts*: $Z \in$ *nxts_rlin2_set P* $\{A\}$ *w* **and** *Z_eps*: $(Z,[]) \in P$
unfolding *accepted_def* **by** *blast*
from *Z_nxts* **obtain** *B* **where** *B_der*: $P \vdash [Nt\ B] \Rightarrow* map\ Tm\ w\ @\ [Nt\ Z]$ **and** *B_in*: $B \in \{A\}$
using *nxts_to_mult_derive* **by** *fast*
from *B_in* **have** *A_eq_B*: $A = B$ **by** *simp*
from *Z_eps* **have** $P \vdash [Nt\ Z] \Rightarrow []$
using *derive_singleton*[*of P Nt Z* []] **by** *simp*
hence $P \vdash map\ Tm\ w\ @\ [Nt\ Z] \Rightarrow map\ Tm\ w$
using *derive_prepend*[*of P* [*Nt Z*] [] *map Tm w*] **by** *simp*
with *B_der A_eq_B* **have** $P \vdash [Nt\ A] \Rightarrow* map\ Tm\ w$ **by** *simp*
thus *?thesis*
unfolding *Lang_def* **by** *blast*
qed

theorem *Lang_iff_accepted_if_rlin2*:
assumes *rlin2 P*
shows $w \in$ *Lang P A* $\longleftrightarrow$ *accepted P A w*
using *accepted_if_Lang*[*OF assms*] *Lang_if_accepted* **by** *fast*

end

16 Relating Strongly Right-Linear Grammars and Automata

theory *Right_Linear_Automata*
imports
NDA_rlin2
Finite_Automata_HF.Finite_Automata_HF
HereditarilyFinite.Finitary
begin

16.1 From Strongly Right-Linear Grammar to NFA

definition *nfa_rlin2* :: $('n,'t)Prods \Rightarrow 'n \Rightarrow ('t,'n)$ *nfa* **where**
nfa_rlin2 P S =
(| *states* = $\{S\} \cup$ *Nts P*,
init = $\{S\}$,
final = $\{A \in$ *Nts P*. $(A,[]) \in P\}$,
nxt = $\lambda q\ a.$ *nxt_rlin2 P q a*,
eps = *Id* |)

context
fixes *P* :: $('n,'t)Prods$
assumes *finite P*
begin

interpretation *NFA_rlin2*: *nfa nfa_rlin2 P S*
unfolding *nfa_rlin2_def* **proof** (*standard*, *goal_cases*)
case *1*
then show *?case* **by**(*simp*)
next
case *2*
then show *?case* **by** *auto*
next
case (*3 q x*)
then show *?case* **by**(*auto simp add*: *nxt_rlin2_nts*)
next
case *4*
then show *?case* **using** ‹*finite P*› **by** (*simp add*: *Nts_def finite_nts_syms split_def*)
qed
print_theorems

lemma *nfa_init_nfa_rlin2*: *nfa.init* (*nfa_rlin2 P S*) = $\{S\}$
by (*simp add*: *nfa_rlin2_def*)

lemma *nfa_final_nfa_rlin2*: *nfa.final* (*nfa_rlin2 P S*) = $\{A \in$ *Nts P*. $(A,[]) \in P\}$
by (*simp add*: *nfa_rlin2_def*)

lemma *nfa_nxt_nfa_rlin2*: *nfa.nxt* (*nfa_rlin2 P S*) *A a* = *nxt_rlin2 P A a*
by (*simp add*: *nfa_rlin2_def*)

lemma *nfa_epsclo_nfa_rlin2*: *M* ⊆ {*S*} ∪ *Nts P* ⟹ *nfa.epsclo* (*nfa_rlin2 P S*)
M = *M*
unfolding *NFA_rlin2.epsclo_def* **unfolding** *nfa_rlin2_def* **by**(*auto*)

lemma *nfa_nextl_nfa_rlin2*: *M* ⊆ {*S*} ∪ *Nts P*
⟹ *nfa.nextl* (*nfa_rlin2 P S*) *M xs* = *nxts_rlin2_set P M xs*
proof(*induction xs arbitrary*: *M*)
case *Nil*
then show *?case*
by (*simp add*: *nxts_rlin2_set_def*)(*fastforce intro*!: *nfa_epsclo_nfa_rlin2*)
next
case (*Cons a xs*)
let *?epsclo* = *nfa.epsclo* (*nfa_rlin2 P S*)
let *?nxt* = *nfa.nxt* (*nfa_rlin2 P S*)
let *?nxts* = *nfa.nextl* (*nfa_rlin2 P S*)
have *?nxts M* (*a* # *xs*) = *?nxts* (⋃*x*∈*?epsclo M*. *?nxt x a*) *xs*
by *simp*
also have ... = *?nxts* (⋃*x*∈*M*. *?nxt x a*) *xs*
using *Cons.prems* **by**(*subst nfa_epsclo_nfa_rlin2*) *auto*
also have ... = *?nxts* (⋃*m*∈*M*. *nxt_rlin2 P m a*) *xs*
by (*simp add*: *nfa_nxt_nfa_rlin2*)
also have ... = *nxts_rlin2_set P* (⋃*m*∈*M*. *nxt_rlin2 P m a*) *xs*
using *Cons.prems* **by**(*subst Cons.IH*)(*auto simp add*: *nxt_rlin2_nts*)
also have ... = *nxts_rlin2_set P M* (*a* # *xs*)
by (*simp add*: *nxt_rlin2_set_def nxts_rlin2_set_def*)
finally show *?case* **.**
qed

lemma *lang_pres_nfa_rlin2*: **assumes** *rlin2 P*
shows *nfa.language* (*nfa_rlin2 P S*) = *Lang P S*
proof −
have *1*: ⋀*A xs*. ⟦ *A* ∈ *nxts_rlin2_set P* {*S*} *xs*; *A* ∈ *Nts P*; (*A*, []) ∈ *P*⟧ ⟹
P ⊢ [*Nt S*] ⇒∗ *map Tm xs*
using *nxts_to_mult_derive* **by** (*metis* (*no_types*, *opaque_lifting*) *append.right_neutral*
derive.intros
r_into_rtranclp rtranclp_trans singletonD)
have ⋀*A B*. *Nt B* ∉ *Syms P* ⟹ (*A*, []) ∈ *P* ⟹ *A* ≠ *B* **by**(*auto simp*: *Syms_def*)
hence *2*: ⋀*xs*. *rlin2 P* ⟹ *P* ⊢ [*Nt S*] ⇒∗ *map Tm xs* ⟹
nxts_rlin2_set P {*S*} *xs* ∩ {*A* ∈ *Nts P*. (*A*, []) ∈ *P*} ≠ {}
using *in_Nts_iff_in_Syms mult_derive_to_nxts rlin2_tms_eps*
by (*metis* (*no_types*, *lifting*) *Int_Collect empty_iff singletonI*)
show *?thesis*
unfolding *NFA_rlin2.language_def Lang_def nfa_init_nfa_rlin2 nfa_final_nfa_rlin2*
nfa_nextl_nfa_rlin2[*OF Un_upper1*]
using *2*[*OF assms*] **by** (*auto simp*: *1*)
qed

lemma *regular_if_rlin2*: **assumes** *rlin2 P*
shows *regular* (*Lang P S*)
using *lang_pres_nfa_rlin2*[*OF assms*] *NFA_rlin2.imp_regular*[*of S*]
by *metis*

end

16.2 From DFA to Strongly Right-Linear Grammar

context *dfa*
begin

We define *Prods_dfa* that collects the production set from the deterministic finite automata *M*

definition *Prods_dfa* :: ('*s*, '*a*) *Prods* **where**
Prods_dfa =
(⋃ *q* ∈ *dfa.states M*. ⋃ *x*. {(*q*,[*Tm x*, *Nt*(*dfa.nxt M q x*)])}) ∪ (⋃ *q* ∈ *dfa.final M*. {(*q*,[])})

lemma *rlin2_prods_dfa*: *rlin2* (*Prods_dfa*)
unfolding *rlin2_def Prods_dfa_def* **by** *blast*

We show that a word can be derived from the production set *Prods_dfa* if and only if traversing the word in the deterministic finite automata *M* ends in a final state. The proofs are very similar to those in *DFA_rlin2.thy*

lemma *mult_derive_to_nxtl*:
Prods_dfa ⊢ [*Nt A*] ⇒∗ *map Tm w* @ [*Nt B*] ⟹ *nextl A w* = *B*
proof (*induction w arbitrary*: *B rule*: *rev_induct*)
case *Nil*
thus *?case*
using *rlin2_nts_derive_eq*[*OF rlin2_prods_dfa*, *of A B*] **by** *simp*
next
case (*snoc x xs*)
from *snoc.prems* **have** *Prods_dfa* ⊢ [*Nt A*] ⇒∗ *map Tm xs* @ [*Tm x*,*Nt B*] **by** *simp*
then obtain *C* **where** *C_der*: *Prods_dfa* ⊢ [*Nt A*] ⇒∗ *map Tm xs* @ [*Nt C*]
and *C_prods*: (*C*,[*Tm x*, *Nt B*]) ∈ *Prods_dfa* **using** *rlin2_introduce_tm*[*OF rlin2_prods_dfa*, *of A xs x B*] **by** *auto*
have *1*: *nextl A xs* = *C*
using *snoc.IH*[*OF C_der*] **.**
from *C_prods* **have** *2*: *B* = *dfa.nxt M C x*
unfolding *Prods_dfa_def* **by** *blast*
from *1 2* **show** *?case* **by** *simp*
qed

lemma *nxtl_to_mult_derive*:
assumes *A* ∈ *dfa.states M*
shows *Prods_dfa* ⊢ [*Nt A*] ⇒∗ *map Tm w* @ [*Nt* (*nextl A w*)]

```
proof (induction w rule: rev_induct)
  case (snoc x xs)
  let ?B = nextl A xs
  have ?B ∈ dfa.states M
    using nextl_state[OF assms, of xs] .
  hence (?B, [Tm x, Nt (dfa.nxt M ?B x)]) ∈ Prods_dfa
    unfolding Prods_dfa_def by blast
  hence Prods_dfa ⊢ [Nt ?B] ⇒ [Tm x] @ [Nt (dfa.nxt M ?B x)]
    by (simp add: derive_singleton)
  hence Prods_dfa ⊢ [Nt A] ⇒* map Tm xs @ ([Tm x] @ [Nt (dfa.nxt M ?B x)])
    using snoc.IH by (meson derive_prepend rtranclp.simps)
  thus ?case by auto
qed simp

theorem Prods_dfa_iff_dfa:
  q ∈ dfa.states M ⟹ Prods_dfa ⊢ [Nt q] ⇒* map Tm w ⟷ nextl q w ∈ dfa.final
M
proof
  show Prods_dfa ⊢ [Nt q] ⇒* map Tm w ⟹ nextl q w ∈ dfa.final M
  proof −
    assume asm: Prods_dfa ⊢ [Nt q] ⇒* map Tm w
     obtain B where q_der: Prods_dfa ⊢ [Nt q] ⇒* map Tm w @ [Nt B] and
B_in: (B,[]) ∈ Prods_dfa
     unfolding Lang_def using rlin2_tms_eps[OF rlin2_prods_dfa asm] by auto
    have 1: nextl q w = B
      using mult_derive_to_nxtl[OF q_der] .
    from B_in have 2: B ∈ dfa.final M
      unfolding Prods_dfa_def by blast
    from 1 2 show ?thesis by simp
  qed
next
  assume asm1: q ∈ dfa.states M
  show nextl q w ∈ dfa.final M ⟹ Prods_dfa ⊢ [Nt q] ⇒* map Tm w
  proof −
    assume asm2: nextl q w ∈ dfa.final M
    let ?Z = nextl q w
    from asm2 have Z_eps: (?Z,[]) ∈ Prods_dfa
      unfolding Prods_dfa_def by blast
    have Prods_dfa ⊢ [Nt q] ⇒* map Tm w @ [Nt ?Z]
      using nxtl_to_mult_derive[OF asm1, of w] .
    with Z_eps show ?thesis
      by (metis derives_rule rtranclp.rtrancl_refl self_append_conv)
  qed
qed

corollary dfa_language_eq_Lang: dfa.language M = Lang Prods_dfa (dfa.init M)
unfolding language_def Lang_def by (simp add: Prods_dfa_iff_dfa)

end
```

corollary *rlin2_if_regular*:
regular L $\Longrightarrow \exists$ *P S*::*hf*. *rlin2 P* $\wedge$ *L* = *Lang P S*
by (*metis dfa.dfa_language_eq_Lang dfa.rlin2_prods_dfa regular_def*)

end

17 Pumping Lemma for Strongly Right-Linear Grammars

theory *Pumping_Lemma_Regular*
imports *NDA_rlin2 List_Power.List_Power*
begin

The proof is on the level of strongly right-linear grammars. Currently there is no proof on the automaton level but now it would be easy to obtain one.

lemma *not_distinct*:
assumes *m* = *card P*
and *m* $\geq$ *1*
and $\forall$ *i* < *length w*. *w* ! *i* $\in$ *P*
and *length w* $\geq$ *Suc m*
shows $\exists$ *xs ys zs y*. *w* = *xs* @ [*y*] @ *ys* @ [*y*] @ *zs* $\wedge$ *length* (*xs* @ [*y*] @ *ys* @ [*y*]) $\leq$ *Suc m*
using *assms* **proof** (*induction w arbitrary*: *P m rule*: *length_induct*)
case (*1 aw*)
from *1.prems*(*4*) **obtain** *a w* **where** *aw_cons*[*simp*]: *aw* = *a*#*w* **and** *w_len*: *m* $\leq$ *length w*
using *Suc_le_length_iff*[*of m aw*] **by** *blast*
show *?case* **proof** (*cases a* $\in$ *set w*)
case *True*
hence $\neg$ *distinct aw* **by** *simp*
then obtain *xs ys zs y* **where** *aw_split*: *aw* = *xs* @ [*y*] @ *ys* @ [*y*] @ *zs*
using *not_distinct_decomp* **by** *blast*
show *?thesis* **proof** (*cases length* (*xs* @ [*y*] @ *ys* @ [*y*]) $\leq$ *Suc m*)
case *True*
with *aw_split* **show** *?thesis* **by** *blast*
next
case *False*
let *?xsyys* = *xs* @ [*y*] @ *ys*
from *False* **have** *a4*: *length ?xsyys* $\geq$ *Suc m* **by** *simp*
from *aw_split* **have** *a5*: *length ?xsyys* < *length aw* **by** *simp*
with *1.prems*(*3*) **have** $\forall$ *i*<*length ?xsyys*. *aw* ! *i* $\in$ *P* **by** *simp*
with *aw_split* **have** *a3*: $\forall$ *i* < *length ?xsyys*. *?xsyys* ! *i* $\in$ *P*
by (*metis append_assoc nth_append*)
from *1.prems*(*2*) *1.prems*(*1*) *a3 a4 a5* **have** $\exists$ *xs' ys' zs' y'*. *?xsyys* = *xs'* @ [*y'*] @ *ys'* @ [*y'*] @ *zs'* $\wedge$ *length* (*xs'* @ [*y'*] @ *ys'* @ [*y'*]) $\leq$ *Suc m*
using *1.IH* **by** *simp*

then obtain *xs′ ys′ zs′ y′* **where** *xsyys_split*: *?xsyys = xs′ @ [y′] @ ys′ @ [y′] @ zs′* **and** *xsyys′_len*: *length (xs′ @ [y′] @ ys′ @ [y′]) ≤ Suc m* **by** *blast*
let *?xs = xs′* **let** *?y = y′* **let** *?ys = ys′* **let** *?zs = zs′ @ [y] @ zs*
from *xsyys_split aw_split* **have** ∗: *aw = ?xs @ [?y] @ ?ys @ [?y] @ ?zs* **by** *simp*
from *xsyys′_len* **have** ∗∗: *length (?xs @ [?y] @ ?ys @ [?y]) ≤ Suc m* **by** *simp*
from ∗ ∗∗ **show** *?thesis* **by** *blast*
qed
next
case *False*
let *?P′ = P − {a}*
from *1.prems(3)* **have** *a_in*: *a ∈ P* **by** *auto*
with *1.prems(1)* **have** *a1*: *m−1 = card ?P′* **by** *simp*
from *1.prems(2) w_len* **have** *w ≠ []* **by** *auto*
with *1.prems(3) False* **have** *b_in*: *∃ b≠a. b ∈ P* **by** *force*
from *a_in b_in 1.prems(2) 1.prems(1)* **have** *m ≥ 2*
by (*metis Suc_1 card_1_singletonE not_less_eq_eq singletonD verit_la_disequality*)
hence *a2*: *m−1 ≥ 1* **by** *simp*
from *False 1.prems(3)* **have** *a3*: *∀ i<length w. w ! i ∈ ?P′*
using *DiffD2* **by** *auto*
from *1.prems(2) w_len* **have** *a4*: *Suc (m−1) ≤ length w* **by** *simp*
from *a1 a2 a3 a4* **have** *∃ xs ys zs y. w = xs @ [y] @ ys @ [y] @ zs ∧ length (xs @ [y] @ ys @ [y]) ≤ Suc (m − 1)*
using *1.IH* **by** *simp*
then obtain *xs ys zs y* **where** *w_split*: *w = xs @ [y] @ ys @ [y] @ zs* **and** *xsys_len*: *length (xs @ [y] @ ys @ [y]) ≤ m* **by** *auto*
from *w_split* **have** ∗: *a#w = (a#xs) @ [y] @ ys @ [y] @ zs* **by** *simp*
from *xsys_len* **have** ∗∗: *length ((a#xs) @ [y] @ ys @ [y]) ≤ Suc m* **by** *simp*
from ∗ ∗∗ *aw_cons* **show** *?thesis* **by** *blast*
qed
qed

We define the function *nxts_nts P a w* that collects all paths traversing the word *w* starting from the non-terminal *A* in the production set *P*. *nxts_nts0* appends the non-terminal *A* in front of every list produced by *nxts_nts*

fun *nxts_nts* :: (*'n,'t*)*Prods* ⇒ *'n* ⇒ *'t list* ⇒ *'n list set* **where**
nxts_nts P A [] = {[]}
| *nxts_nts P A (a#w) = (⋃ B∈nxt_rlin2 P A a. (Cons B)‘nxts_nts P B w)*

definition *nxts_nts0* **where**
nxts_nts0 P A w ≡ ((#) A) ‘ nxts_nts P A w

17.1 Properties of *nxts_nts* and *nxts_nts0*

lemma *nxts_nts0_i0*:
∀ e ∈ nxts_nts0 P A w. e!0 = A
unfolding *nxts_nts0_def* **by** *auto*

lemma *nxts_nts0_shift*:
assumes $i < length\ w$
shows $\forall e \in nxts\_nts0\ P\ A\ w.\ \exists e' \in nxts\_nts\ P\ A\ w.\ e\ !\ (Suc\ i) = e'\ !\ i$
unfolding *nxts_nts0_def* **by** *auto*

lemma *nxts_nts_pick_nt*:
assumes $e \in nxts\_nts\ P\ A\ (a\#w)$
shows $\exists C{\in}nxt\_rlin2\ P\ A\ a.\ \exists e' \in nxts\_nts\ P\ C\ w.\ e = C\#e'$
using *assms* **by** *auto*

lemma *nxts_nts0_len*:
$\forall e \in nxts\_nts0\ P\ A\ w.\ length\ e = Suc\ (length\ w)$
unfolding *nxts_nts0_def*
by (*induction P A w rule*: *nxts_nts.induct*) *auto*

lemma *nxts_nts0_nxt*:
assumes $i < length\ w$
shows $\forall e \in nxts\_nts0\ P\ A\ w.\ e!(Suc\ i) \in nxt\_rlin2\ P\ (e!i)\ (w!i)$
unfolding *nxts_nts0_def* **using** *assms* **proof** (*induction P A w arbitrary*: *i rule*: *nxts_nts.induct*)
case (*1 P A*)
thus *?case* **by** *simp*
next
case (*2 P A a w*)
thus *?case*
using *less_Suc_eq_0_disj* **by** *auto*
qed

lemma *nxts_nts0_path*:
assumes $i1 \leq length\ w$
and $i2 \leq length\ w$
and $i1 \leq i2$
shows $\forall e \in nxts\_nts0\ P\ A\ w.\ e!i2 \in nxts\_rlin2\_set\ P\ \{e!i1\}\ (drop\ i1\ (take\ i2\ w))$
proof
fix *e*
assume $e \in nxts\_nts0\ P\ A\ w$
with *assms* **show** $e\ !\ i2 \in nxts\_rlin2\_set\ P\ \{e\ !\ i1\}\ (drop\ i1\ (take\ i2\ w))$ **proof** (*induction i2−i1 arbitrary*: *i2*)
case *0*
thus *?case*
by (*simp add*: *nxts_rlin2_set_def*)
next
case (*Suc x*)
let $?i2' = i2 - 1$
from *Suc.hyps*(*2*) **have** *x_def*: $x = ?i2' - i1$ **by** *simp*
from *Suc.prems*(*2*) **have** *i2'_len*: $?i2' \leq length\ w$ **by** *simp*
from *Suc.prems*(*3*) *Suc.hyps*(*2*) **have** *i1_i2'*: $i1 \leq ?i2'$ **by** *simp*
have *IH*: $e\ !\ ?i2' \in nxts\_rlin2\_set\ P\ \{e\ !\ i1\}\ (drop\ i1\ (take\ ?i2'\ w))$

```
      using Suc.hyps(1)[of ?i2', OF x_def Suc.prems(1) i2'_len i1_i2' Suc.prems(4)]
.
       from Suc.hyps(2) Suc.prems(2) Suc.prems(4) have e ! i2 ∈ nxt_rlin2 P
(e!(i2−1)) (w!(i2−1))
        using nxts_nts0_nxt[of ?i2' w P A] by simp
      hence e_i2: e ! i2 ∈ nxts_rlin2_set P {e!(i2−1)} [w!(i2−1)]
        unfolding nxts_rlin2_set_def nxt_rlin2_set_def by simp
      have drop i1 (take (i2 − 1) w) @ [w ! (i2 − 1)] = drop i1 (take i2 w)
      by (smt (verit) Cons_nth_drop_Suc Suc.hyps(2) Suc.prems(2) Suc.prems(3)
add_Suc drop_drop drop_eq_Nil drop_take i1_i2' i2'_len le_add_diff_inverse2
le_less_Suc_eq nle_le nth_via_drop order.strict_iff_not take_Suc_conv_app_nth
x_def)
      thus ?case
        using nxts_trans2[of e ! (i2 − 1) P e ! i1 drop i1 (take (i2 − 1) w) e ! i2
[w!(i2−1)], OF IH e_i2] by argo
  qed
qed

lemma nxts_nts0_path_start:
  assumes i ≤ length w
  shows ∀ e ∈ nxts_nts0 P A w. e ! i ∈ nxts_rlin2_set P {A} (take i w)
  using assms nxts_nts0_path[of 0 w i P A] by (simp add: nxts_nts0_def)

lemma nxts_nts_elem:
  assumes i < length w
  shows ∀ e ∈ nxts_nts P A w. e ! i ∈ Nts P
proof
  fix e
  assume e ∈ nxts_nts P A w
   with assms show e ! i ∈ Nts P proof (induction P A w arbitrary: i e rule:
nxts_nts.induct)
    case (1 P A)
    thus ?case by simp
  next
    case (2 P A a w)
    from 2(3) obtain C e' where C_def: C ∈ nxt_rlin2 P A a and e'_def: e'
∈ nxts_nts P C w and e_app: e = C#e'
      using nxts_nts_pick_nt[of e P A a w] by blast
    show ?case proof (cases i = 0)
      case True
      with e_app C_def show ?thesis
        using nxt_rlin2_nts by simp
    next
      case False
      from False 2(2) have i_len: i − 1 < length w by simp
      have e' ! (i − 1) ∈ Nts P
        using 2.IH[of C i−1 e', OF C_def i_len e'_def] .
      with e_app False have e ! i ∈ Nts P by simp
      thus ?thesis .
```

```
    qed
  qed
qed

lemma nxts_nts0_elem:
  assumes A ∈ Nts P
      and i ≤ length w
  shows ∀ e ∈ nxts_nts0 P A w. e ! i ∈ Nts P
proof (cases i = 0)
  case True
  thus ?thesis
    by (simp add: assms(1) nxts_nts0_i0)
next
  case False
  show ?thesis proof
    fix e
    assume e_def: e ∈ nxts_nts0 P A w
    from False e_def assms(2) have ∃ e' ∈ nxts_nts P A w. e ! i = e' ! (i−1)
      using nxts_nts0_shift[of i−1 w P A] by simp
    then obtain e' where e'_def: e' ∈ nxts_nts P A w and e_ind: e ! i = e' !
(i−1)
      by blast
    from False e'_def assms(2) have e' ! (i−1) ∈ Nts P
      using nxts_nts_elem[of i−1 w P A] by simp
    with e_ind show e ! i ∈ Nts P by simp
  qed
qed

lemma nxts_nts0_pick:
  assumes B ∈ nxts_rlin2_set P {A} w
  shows ∃ e ∈ nxts_nts0 P A w. last e = B
unfolding nxts_nts0_def using assms proof (induction P A w arbitrary: B rule:
nxts_nts.induct)
  case (1 P A)
  thus ?case
    by (simp add: nxts_rlin2_set_def)
next
  case (2 P A a w)
  from 2(2) obtain C where C_def: C ∈ nxt_rlin2 P A a and C_path: B ∈
nxts_rlin2_set P {C} w
    using nxts_rlin2_set_first_step[of B P A a w] by blast
  have ∃ e ∈ nxts_nts0 P C w. last e = B
    using 2.IH[of C B, OF C_def C_path] by (simp add: nxts_nts0_def)
  then obtain e where e_def: e ∈ nxts_nts0 P C w and e_last: last e = B
    by blast
  from e_def C_def have ∗: A#e ∈ nxts_nts0 P A (a#w)
    unfolding nxts_nts0_def by auto
  from e_last e_def have ∗∗: last (A#e) = B
    using nxts_nts0_len[of P C w] by auto
```

from $*$ $**$ **show** *?case*
unfolding *nxts_nts0_def* **by** *blast*
qed

17.2 Pumping Lemma

The following lemma states that in the automata level there exists a cycle occurring in the first m symbols where m is the cardinality of the non-terminals set, under the following assumptions

lemma *nxts_split_cycle*:
assumes *finite P*
and $A \in Nts\ P$
and $m = card\ (Nts\ P)$
and $B \in nxts\_rlin2\_set\ P\ \{A\}\ w$
and $length\ w \geq m$
shows $\exists x\ y\ z\ C.\ w = x@y@z \wedge length\ y \geq 1 \wedge length\ (x@y) \leq m \wedge$
$C \in nxts\_rlin2\_set\ P\ \{A\}\ x \wedge C \in nxts\_rlin2\_set\ P\ \{C\}\ y \wedge B \in nxts\_rlin2\_set\ P\ \{C\}\ z$
proof −
let *?nts* = *nxts_nts0 P A w*
obtain *e* **where** *e_def*: $e \in$ *?nts* **and** *e_last*: *last e* = *B*
using *nxts_nts0_pick*[*of B P A w*, *OF assms*(*4*)] **by** *auto*
from *e_def* **have** *e_len*: *length e* = *Suc* (*length w*)
using *nxts_nts0_len*[*of P A w*] **by** *simp*
from *e_len e_def* **have** *e_elem*: $\forall i < length\ e.\ e!i \in Nts\ P$
using *nxts_nts0_elem*[*OF assms*(*2*)] **by** (*auto simp*: *less_Suc_eq_le*)
have *finite* (*Nts P*)
using *finite_Nts*[*of P*, *OF assms*(*1*)] **.**
with *assms*(*2*) *assms*(*3*) **have** *m_geq_1*: $m \geq 1$
using *less_eq_Suc_le* **by** *fastforce*
from *assms*(*5*) *e_len* **have** $\exists xs\ ys\ zs\ y.\ e = xs\ @\ [y]\ @\ ys\ @\ [y]\ @\ zs \wedge length\ (xs\ @\ [y]\ @\ ys\ @\ [y]) \leq Suc\ m$
using *not_distinct*[*OF assms*(*3*) *m_geq_1 e_elem*] **by** *simp*
then obtain *xs ys zs C* **where** *e_split*: *e* = *xs* @ [*C*] @ *ys* @ [*C*] @ *zs* **and** *xy_len*: *length* (*xs* @ [*C*] @ *ys* @ [*C*]) ≤ *Suc m*
by *blast*
let *?e1* = *xs* @ [*C*] **let** *?e2* = *ys* @ [*C*] **let** *?e3* = *zs*
let *?x* = *take* (*length ?e1* − *1*) *w* **let** *?y* = *drop* (*length ?e1* − *1*) (*take* (*length ?e1*+*length ?e2* − *1*) *w*)
let *?z* = *drop* (*length ?e1*+*length ?e2* − *1*) *w*
have $*$: *w* = *?x*@*?y*@*?z*
by (*metis Nat.add_diff_assoc2 append_assoc append_take_drop_id diff_add_inverse drop_take le_add1 length_append_singleton plus_1_eq_Suc take_add*)
from *e_len e_split* **have** $**$: *length ?y* ≥ *1* **by** *simp*
from *xy_len* **have** $***$: *length* (*?x*@*?y*) ≤ *m* **by** *simp*
have *x_fac*: *?x* = *take* (*length xs*) *w* **by** *simp*
from $**$ **have** *x_fac2*: *length xs* ≤ *length w* **by** *simp*
from *e_split* **have** *x_fac3*: *e* ! *length xs* = *C* **by** *simp*
from *e_def x_fac x_fac3* **have** $****$: *C* ∈ *nxts_rlin2_set P* {*A*} *?x*

```
    using nxts_nts0_path_start[of length xs w P A, OF x_fac2] by auto
  have y_fac: ?y = drop (length xs) (take (length xs + length ys + 1) w) by simp
  from e_len e_split have y_fac2: length xs + length ys + 1 ≤ length w by simp
  have y_fac3: length xs ≤ length xs + length ys + 1 by simp
  have y_fac4: e ! (length xs + length ys + 1) = C
    by (metis add.right_neutral add_Suc_right append.assoc append_Cons e_split
length_Cons length_append list.size(3) nth_append_length plus_1_eq_Suc)
  from e_def y_fac x_fac3 y_fac4 have *****: C ∈ nxts_rlin2_set P {C} ?y
    using nxts_nts0_path[of length xs w length xs + length ys + 1 P A, OF x_fac2
y_fac2 y_fac3] by auto
  have z_fac: ?z = drop (length xs + length ys + 1) (take (length w) w) by simp
  from e_last e_len have z_fac2: e ! (length w) = B
    by (metis Zero_not_Suc diff_Suc_1 last_conv_nth list.size(3))
  from e_def z_fac y_fac2 y_fac4 z_fac2 have ******: B ∈ nxts_rlin2_set P
{C} ?z
    using nxts_nts0_path[of length xs + length ys + 1 w length w P A] by auto
  from * ** *** **** ***** ****** show ?thesis by blast
qed
```

We also show that a cycle can be pumped in the automata level

```
lemma pump_cycle:
  assumes B ∈ nxts_rlin2_set P {A} x
      and B ∈ nxts_rlin2_set P {B} y
    shows B ∈ nxts_rlin2_set P {A} (x@(y^^i))
using assms proof (induction i)
  case 0
  thus ?case by (simp add: assms(1))
next
  case (Suc i)
  have B ∈ nxts_rlin2_set P {A} (x@(y^^i))
    using Suc.IH[OF assms] .
  with assms(2) have B ∈ nxts_rlin2_set P {A} (x@(y^^i)@y)
    using nxts_trans2[of B P A x@(y^^i) B y] by simp
  thus ?case
    by (simp add: pow_list_comm)
qed
```

Combining the previous lemmas we can prove the pumping lemma where the starting non-terminal is in the production set. We simply extend the lemma for non-terminals that are not part of the production set, as these non-terminals will produce the empty language

```
lemma pumping_re_aux:
  assumes finite P
      and A ∈ Nts P
      and m = card (Nts P)
      and accepted P A w
      and length w ≥ m
    shows ∃ x y z. w = x@y@z ∧ length y ≥ 1 ∧ length (x@y) ≤ m ∧ (∀ i. accepted
P A (x@(y^^i)@z))
```

proof −
from *assms*(*4*) **obtain** *Z* **where** *Z_in*:*Z* ∈ *nxts_rlin2_set P* {*A*} *w* **and** *Z_eps*:(*Z*,[])∈*P*
by (*auto simp*: *accepted_def*)
obtain *x y z C* **where** ∗: *w* = *x*@*y*@*z* **and** ∗∗: *length y* ≥ *1* **and** ∗∗∗: *length* (*x*@*y*) ≤ *m* **and**
1: *C* ∈ *nxts_rlin2_set P* {*A*} *x* **and** *2*: *C* ∈ *nxts_rlin2_set P* {*C*} *y* **and** *3*: *Z* ∈ *nxts_rlin2_set P* {*C*} *z*
using *nxts_split_cycle*[*OF assms*(*1*) *assms*(*2*) *assms*(*3*) *Z_in assms*(*5*)] **by** *auto*
have ∀ *i*. *C* ∈ *nxts_rlin2_set P* {*A*} (*x*@(*y*^^*i*))
using *pump_cycle*[*OF 1 2*] **by** *simp*
with *3* **have** ∀ *i*. *Z* ∈ *nxts_rlin2_set P* {*A*} (*x*@(*y*^^*i*)@*z*)
using *nxts_trans2*[*of C P A*] **by** *fastforce*
with *Z_eps* **have** ∗∗∗∗: (∀ *i*. *accepted P A* (*x*@(*y*^^*i*)@*z*))
by (*auto simp*: *accepted_def*)
from ∗ ∗∗ ∗∗∗ ∗∗∗∗ **show** *?thesis* **by** *auto*
qed

theorem *pumping_lemma_re_nts*:
assumes *rlin2 P*
and *finite P*
and *A* ∈ *Nts P*
shows ∃ *n*. ∀ *w* ∈ *Lang P A*. *length w* ≥ *n* ⟶
(∃ *x y z*. *w* = *x*@*y*@*z* ∧ *length y* ≥ *1* ∧ *length* (*x*@*y*) ≤ *n* ∧ (∀ *i*. *x*@(*y*^^*i*)@*z* ∈ *Lang P A*))
using *assms pumping_re_aux*[*of P A card* (*Nts P*)] *Lang_iff_accepted_if_rlin2*[*OF assms*(*1*)] **by** *metis*

theorem *pumping_lemma_regular*:
assumes *rlin2 P* **and** *finite P*
shows ∃ *n*. ∀ *w* ∈ *Lang P A*. *length w* ≥ *n* ⟶
(∃ *x y z*. *w* = *x*@*y*@*z* ∧ *length y* ≥ *1* ∧ *length* (*x*@*y*) ≤ *n* ∧ (∀ *i*. *x*@(*y*^^*i*)@*z* ∈ *Lang P A*))
proof (*cases A* ∈ *Nts P*)
case *True*
thus *?thesis*
using *pumping_lemma_re_nts*[*OF assms True*] **by** *simp*
next
case *False*
hence *Lang P A* = {}
by (*auto intro*!: *Lang_empty_if_notin_Lhss simp add*: *Lhss_def Nts_def*)
thus *?thesis* **by** *simp*
qed

Most of the time pumping lemma is used in the contrapositive form to prove that no right-linear set of productions exists.

corollary *pumping_lemma_regular_contr*:
assumes *finite P*
and ∀ *n*. ∃ *w* ∈ *Lang P A*. *length w* ≥ *n* ∧ (∀ *x y z*. *w* = *x*@*y*@*z* ∧ *length y* ≥

1 ∧ *length* (*x*@*y*) ≤ *n* ⟶ (∃ *i*. *x*@(*y*^^*i*)@*z* ∉ *Lang P A*))
shows ¬*rlin2 P*
using *assms pumping_lemma_regular*[*of P A*] **by** *metis*

end

18 $a^n b^n$ is Not Regular

theory *AnBn_Not_Regular*
imports *Pumping_Lemma_Regular*
begin

The following theorem proves that the language *a^nb^n* cannot be produced by a right linear production set, using the contrapositive form of the pumping lemma

theorem *not_rlin2_ab*:
assumes *a* ≠ *b*
and *Lang P A* = (⋃ *n*. {[*a*]^^*n* @ [*b*]^^*n*}) (**is** _ = *?AnBn*)
and *finite P*
shows ¬*rlin2 P*
proof −
have ∃ *w* ∈ *Lang P A*. *length w* ≥ *n* ∧ (∀ *x y z*. *w* = *x*@*y*@*z* ∧ *length y* ≥ *1* ∧ *length* (*x*@*y*) ≤ *n* ⟶ (∃ *i*. *x*@(*y*^^*i*)@*z* ∉ *Lang P A*)) **for** *n*
proof −
let *?anbn* = [*a*]^^*n* @ [*b*]^^*n*
show *?thesis*
proof
have ∗∗: (∃ *i*. *x* @ (*y*^^*i*) @ *z* ∉ *Lang P A*)
if *asm*: *?anbn* = *x* @ *y* @ *z* ∧ *1* ≤ *length y* ∧ *length* (*x* @ *y*) ≤ *n* **for** *x y z*
proof
from *asm* **have** *asm1*: [*a*]^^*n* @ [*b*]^^*n* = *x* @ *y* @ *z* **by** *blast*
from *asm* **have** *asm2*: *1* ≤ *length y* **by** *blast*
from *asm* **have** *asm3*: *length* (*x* @ *y*) ≤ *n* **by** *blast*
let *?kx* = *length x* **let** *?ky* = *length y*
have *splitted*: *x* = [*a*]^^*?kx* ∧ *y* = [*a*]^^*?ky* ∧ *z* = [*a*]^^(*n* − *?kx* − *?ky*) @ [*b*]^^*n*
proof −
have ∀ *i* < *n*. ([*a*]^^*n* @ [*b*]^^*n*)!*i* = *a*
by (*simp add*: *nth_append nth_pow_list_single*)
with *asm1* **have** *xyz_tma*: ∀ *i* < *n*. (*x*@*y*@*z*)!*i* = *a* **by** *metis*
with *asm3* **have** *xy_tma*: ∀ *i* < *length*(*x*@*y*). (*x*@*y*)!*i* = *a*
by (*metis append_assoc nth_append order_less_le_trans*)
from *xy_tma* **have** ∀ *i* < *?kx*. *x*!*i* = *a*
by (*metis le_add1 length_append nth_append order_less_le_trans*)
hence ∗: *x* = [*a*]^^*?kx*
by (*simp add*: *list_eq_iff_nth_eq nth_pow_list_single*)
from *xy_tma* **have** ∀ *i* < *?ky*. *y*!*i* = *a*
by (*metis length_append nat_add_left_cancel_less nth_append_length_plus*)
hence ∗∗: *y* = [*a*]^^*?ky*

```
          by (simp add: nth_equalityI pow_list_single)
         from * ** asm1 have [a]^^n @ [b]^^n = [a]^^?kx @ [a]^^?ky @ z by
simp
        hence z_rest: [a]^^n @ [b]^^n = [a]^^(?kx+?ky) @ z
          by (simp add: pow_list_add)
        from asm3 have ***: z = [a]^^(n-?kx-?ky) @ [b]^^n
          using pow_list_eq_appends_iff[THEN iffD1, OF _ z_rest] by simp
        from * ** *** show ?thesis by blast
       qed
       from splitted have x @ y^^2 @ z = [a]^^?kx @ ([a]^^?ky)^^2 @ [a]^^(n
- ?kx - ?ky) @ [b]^^n by simp
      also have ... = [a]^^?kx @ [a]^^(?ky*2) @ [a]^^(n - ?kx - ?ky) @ [b]^^n
        by (simp add: pow_list_mult)
       also have ... = [a]^^(?kx + ?ky*2 + (n - ?kx - ?ky)) @ [b]^^n
        by (simp add: pow_list_add)
       also from asm3 have ... = [a]^^(n+?ky) @ [b]^^n
        by (simp add: add.commute)
       finally have wit: x @ y^^2 @ z = [a]^^(n+?ky) @ [b]^^n .
       from asm2 have [a]^^(n + ?ky) @ [b]^^n ∉ ?AnBn
        using ‹a≠b› by auto
       with wit have x @ y^^2 @ z ∉ ?AnBn by simp
       thus x @ y^^2 @ z ∉ Lang P A using assms(2) by blast
     qed
     from ** show n ≤ length ?anbn ∧ (∀ x y z. ?anbn = x @ y @ z ∧ 1 ≤ length
y ∧ length (x @ y) ≤ n ⟶ (∃ i. x @ (y^^i) @ z ∉ Lang P A)) by simp
    next
     have ?anbn ∈ ?AnBn by blast
     thus ?anbn ∈ Lang P A
      by (simp add: assms(2))
    qed
  qed
  thus ?thesis
   using pumping_lemma_regular_contr[OF assms(3)] by blast
qed

end
```

References

[1] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 3rd edition, 2006.

[2] M. Ramos. Github repository, 2018. Accessed on 8/5/2025. URL: https://github.com/mvmramos/intersection.

[3] M. V. M. Ramos, J. C. B. Almeida, N. Moreira, and R. J. G. B. de Queiroz. Some applications of the formalization of the pumping

lemma for context-free languages. In B. Accattoli and C. Olarte, editors, *Proceedings of the 13th Workshop on Logical and Semantic Frameworks with Applications, LSFA 2018*, volume 344 of *Electronic Notes in Theoretical Computer Science*, pages 151–167. Elsevier, 2018. URL: https://doi.org/10.1016/j.entcs.2019.07.010.