Context-Free Grammars and Languages

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

February 6, 2026

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
Regular−Sets.Regular_Set
begin

lemma *append_Cons_eq_append_Cons*:
$y' \notin set\ xs \Longrightarrow y \notin set\ xs' \Longrightarrow$
$xs\ @\ y\ \#\ zs = xs'\ @\ y'\ \#\ zs' \longleftrightarrow xs = xs' \wedge y = y' \wedge zs = zs'$
by (*induction xs arbitrary*: *xs'*; *force simp*: *Cons_eq_append_conv*)

lemma *insert_conc*: $insert\ w\ W\ @@\ V = \{w\ @\ v \mid v.\ v \in V\} \cup W\ @@\ V$
by *auto*

lemma *conc_insert*: $W\ @@\ insert\ v\ V = \{w\ @\ v \mid w.\ w \in W\} \cup W\ @@\ V$
by *auto*

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 $('n,'t)\ sym = Nt\ 'n \mid Tm\ 't$

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* ⇒ *bool* **where**
isNt S = (∃ *A*. *S* = *Nt A*)

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

lemma *isNt_simps*[*simp*,*code*]:
‹*isNt* (*Nt A*) = *True*›
‹*isNt* (*Tm a*) = *False*›
by (*simp_all add*: *isNt_def*)

lemma *isTm_simps*[*simp*,*code*]:
‹*isTm* (*Nt A*) = *False*›
‹*isTm* (*Tm a*) = *True*›
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*}

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

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*)

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

lemma *Tms_mono*: *P* ⊆ *P'* ⟹ *Tms P* ⊆ *Tms P'*
unfolding *Tms_def Tms_syms_def* **by** *blast*

definition *nts_syms_acc* :: ('*n*,'*t*)*syms* ⇒ '*n list* ⇒ '*n list* **where**
nts_syms_acc = *foldr* (λ*sy ns*. *case sy of Nt A* ⇒ *List.insert A ns* | *Tm* _ ⇒ *ns*)

definition *nts_syms* :: ('*n*,'*t*)*syms* ⇒ '*n list* **where**
nts_syms sys = *nts_syms_acc sys* []

definition *nts* :: ('*n*,'*t*)*prods* ⇒ '*n list* **where**
nts ps = *foldr* (λ(*A*,*sys*) *ns*. *List.insert A* (*nts_syms_acc sys ns*)) *ps* []

definition *tms_syms_acc* :: (*'n*,*'t*)*syms* ⇒ *'t list* ⇒ *'t list* **where**
tms_syms_acc = *foldr* (λ*sy ts*. *case sy of Tm a* ⇒ *List.insert a ts* | *Nt* _ ⇒ *ts*)

definition *tms_syms* :: (*'n*,*'t*)*syms* ⇒ *'t list* **where**
tms_syms sys = *tms_syms_acc sys* []

definition *tms* :: (*'n*,*'t*)*prods* ⇒ *'t list* **where**
tms ps = *foldr* (λ(_,*sys*). *tms_syms_acc sys*) *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 *Rhss_code*[*code*]: *Rhss P A* = *snd* ‘ {*Aw* ∈ *P*. *fst Aw* = *A*}
by(*auto simp add*: *Rhss_def image_iff*)

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*,*code*]: *Nts_syms* [] = {}
unfolding *Nts_syms_def* **by** *auto*

lemma *Nts_syms_Cons*[*simp*,*code*]: *Nts_syms* (*s*#*ss*) = (*case s of Nt A* ⇒ {*A*}
| _ ⇒ {}) ∪ *Nts_syms ss*
by (*auto simp*: *Nts_syms_def split*: *sym.split*)

lemma *Tms_syms_Nil*[*simp*,*code*]: *Tms_syms* [] = {}
unfolding *Tms_syms_def* **by** *auto*

lemma *Tms_syms_Cons*[*simp*,*code*]: *Tms_syms* (*s*#*ss*) = (*case s of Tm a* ⇒ {*a*} | _ ⇒ {}) ∪ *Tms_syms ss*
by (*auto simp*: *Tms_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 *Tms_syms_append*[*simp*]: *Tms_syms* (*u* @ *v*) = *Tms_syms u* ∪ *Tms_syms v*
by (*auto simp*: *Tms_syms_def*)

lemma *Nts_syms_map_Nt*[*simp*]: *Nts_syms* (*map Nt w*) = *set w*
unfolding *Nts_syms_def* **by** *auto*

lemma *Tms_syms_map_Tm*[*simp*]: *Tms_syms* (*map Tm w*) = *set w*
unfolding *Tms_syms_def* **by** *auto*

lemma *Nts_syms_map_Tm*[*simp*]: *Nts_syms* (*map Tm w*) = {}
unfolding *Nts_syms_def* **by** *auto*

lemma *Tms_syms_map_Nt*[*simp*]: *Tms_syms* (*map Nt w*) = {}
unfolding *Tms_syms_def* **by** *auto*

lemma *Nts_syms_rev*: *Nts_syms* (*rev w*) = *Nts_syms w*
by(*auto simp*: *Nts_syms_def*)

lemma *Tms_syms_rev*: *Tms_syms* (*rev w*) = *Tms_syms w*
by(*auto simp*: *Tms_syms_def*)

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

lemma *Tms_syms_empty_iff*: *Tms_syms w* = {} ⟷ (∃ *u*. *w* = *map Nt 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* $\neq \{\}$ $\Longrightarrow \exists\, u\ A\ v.\ w = map\ Tm\ u$ @ *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* $\in$ *Nts P* $\longleftrightarrow$ *Nt B* $\in$ *Syms P*
unfolding *Nts_def Syms_def Nts_syms_def* **by** (*auto*)

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

lemma *Nts_Un*: *Nts* (*P1* $\cup$ *P2*) = *Nts P1* $\cup$ *Nts P2*
by (*simp add*: *Nts_def*)

lemma *Rhs_Nts_Un*: *Rhs_Nts* (*P* $\cup$ *Q*) = *Rhs_Nts P* $\cup$ *Rhs_Nts Q*
by (*simp add*: *Rhs_Nts_def*)

lemma *Rhss_Un*: *Rhss* (*P* $\cup$ *Q*) *A* = *Rhss P A* $\cup$ *Rhss Q A*
by (*auto simp*: *Rhss_def*)

lemma *Rhss_UN*: *Rhss* ($\bigcup$ *PP*) *A* = $\bigcup$ {*Rhss P A* | *P*. *P* $\in$ *PP*}
by (*auto simp*: *Rhss_def*)

lemma *Rhss_empty*[*simp*]: *Rhss* {} *A* = {}
by (*auto simp*: *Rhss_def*)

lemma *Rhss_insert*: *Rhss* (*insert* (*A*,α) *P*) *B* = (*if A* = *B then insert* α (*Rhss P B*) *else Rhss P B*)
by (*auto simp*: *Rhss_def*)

lemma *Nts_Lhss_Rhs_Nts*: *Nts P* = *Lhss P* $\cup$ *Rhs_Nts P*
unfolding *Nts_def Lhss_def Rhs_Nts_def* **by** *auto*

lemma *Nts_Nts_syms*: *w* $\in$ *Rhss P A* $\Longrightarrow$ *Nts_syms w* $\subseteq$ *Nts P*
unfolding *Rhss_def Nts_def* **by** *blast*

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

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

lemma *in_LhssI*: $(A,\alpha) \in P \Longrightarrow A \in Lhss\ P$
by (*auto simp*: *Lhss_def*)

lemma *Lhss_Collect*: $Lhss\ \{p.\ X\ p\} = \{A.\ \exists \alpha.\ X\ (A,\alpha)\}$
by (*auto simp*: *Lhss_def*)

lemma *in_Rhs_NtsI*: $(A,\alpha) \in P \Longrightarrow B \in Nts\_syms\ \alpha \Longrightarrow B \in Rhs\_Nts\ P$
by (*auto simp*: *Rhs_Nts_def*)

lemma *set_nts_syms*: $set(nts\_syms\_acc\ sys\ ns) = Nts\_syms\ sys \cup set\ ns$
unfolding *nts_syms_acc_def*
by(*induction sys arbitrary*: *ns*) (*auto split*: *sym.split*)

lemma *set_nts*: $set(nts\ ps) = Nts\ (set\ ps)$
by(*induction ps*) (*auto simp*: *nts_def Nts_def set_nts_syms split*: *prod.splits*)

lemma *distinct_nts_syms_acc*: $distinct(nts\_syms\_acc\ sys\ ns) = distinct\ ns$
unfolding *nts_syms_acc_def*
by(*induction sys arbitrary*: *ns*) (*auto split*: *sym.split*)

lemma *distinct_nts_syms*: $distinct(nts\_syms\ sys)$
unfolding *nts_syms_def* **by**(*simp add*: *distinct_nts_syms_acc*)

lemma *distinct_nts*: $distinct(nts\ ps)$
by(*induction ps*) (*auto simp*: *nts_def distinct_nts_syms_acc distinct_nts_syms*)

lemma *set_tms_syms_acc*: $set(tms\_syms\_acc\ sys\ ts) = Tms\_syms\ sys \cup set\ ts$
unfolding *tms_syms_acc_def*
by(*induction sys arbitrary*: *ts*) (*auto split*: *sym.split*)

corollary *set_tms_syms*: $set(tms\_syms\ sys) = Tms\_syms\ sys$
unfolding *tms_syms_def Tms_syms_def set_tms_syms_acc Tms_syms_def* **by** (*auto*)

lemma *set_tms*: $set(tms\ ps) = Tms\ (set\ ps)$
by(*induction ps*) (*auto simp*: *tms_def Tms_def set_tms_syms_acc split*: *prod.splits*)

lemma *distinct_tms_syms_acc*: $distinct(tms\_syms\_acc\ sys\ ts) = distinct\ ts$
unfolding *tms_syms_acc_def*
by(*induction sys arbitrary*: *ts*) (*auto split*: *sym.split*)

lemma *distinct_tms_syms*: $distinct(tms\_syms\ sys)$
unfolding *tms_syms_def* **by**(*simp add*: *distinct_tms_syms_acc*)

lemma *distinct_tms*: $distinct(tms\ ps)$
by(*induction ps*) (*auto simp*: *tms_def distinct_tms_syms_acc split*: *sym.split*)

1.1.1 Finiteness Lemmas

lemma *finite_Nts_syms*: *finite* (*Nts_syms w*)
by (*induction w*) (*auto split*: *sym.split*)

lemma *finite_Tms_syms*: *finite* (*Tms_syms w*)
by (*induction w*) (*auto split*: *sym.split*)

lemma *finite_nts*: *finite*(*Nts* (*set ps*))
unfolding *Nts_def* **by** (*simp add*: *finite_Nts_syms split_def*)

lemma *finite_tms*: *finite*(*Tms* (*set ps*))
unfolding *Tms_def* **by** (*simp add*: *finite_Tms_syms split_def*)

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

lemma *finite_nts_prods_start*: *finite*(*Nts*(*set*(*prods g*)) ∪ {*start g*})
unfolding *Nts_def* **by** (*simp add*: *finite_Nts_syms split_def*)

lemma *fresh_nts_prods_start*: *fresh0*(*Nts*(*set*(*prods g*)) ∪ {*start g*}) ∉ *Nts*(*set*(*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_Tms*: *finite P* ⟹ *finite* (*Tms P*)
unfolding *Tms_def* **by** (*simp add*: *case_prod_beta finite_Tms_syms*)

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

1.2 Derivations and Languages

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

inductive *derive* :: ('*n*,'*t*) *Prods* ⇒ ('*n*,'*t*) *syms* ⇒ ('*n*,'*t*)*syms* ⇒ *bool*
((*2*_ ⊢/ (_ ⇒/ _)) [*50*, *0*, *50*] *50*) **for** *P* **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*]

The *language* of a nonterminal is the set of the terminal words it derives.

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*)

Language is extended over mixed words.

definition *Lang_of* :: ('*n*,'*t*) *Prods* ⇒ ('*n*,'*t*) *syms* ⇒ '*t list set* **where**
Lang_of P α = {*w*. *P* ⊢ *α* ⇒∗ *map Tm w*}

abbreviation *Lang_of_set* :: ('*n*,'*t*) *Prods* ⇒ ('*n*,'*t*) *syms set* ⇒ '*t list set* **where**
Lang_of_set P X ≡ ⋃(*Lang_of P* ‘ *X*)

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*: $\neg\ P \vdash map\ Tm\ as \Rightarrow w$
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*)

lemma *Lang_of_mono*: $P \subseteq P' \Longrightarrow Lang\_of\ P\ w \subseteq Lang\_of\ P'\ w$
using *derives_mono* **by** (*auto simp*: *Lang_of_def*)

lemma *derive_set_subset*:
$P \vdash u \Rightarrow v \Longrightarrow set\ v \subseteq set\ u \cup Syms\ P$
by (*auto simp*: *derive_iff Syms_def*)

lemma *deriven_set_subset*:
$P \vdash u \Rightarrow(n)\ v \Longrightarrow set\ v \subseteq set\ u \cup Syms\ P$

by (*induction n arbitrary*: *u*)
(*auto simp*: *relpowp_Suc_left dest*!: *derive_set_subset*)

lemma *derives_set_subset*:
$P \vdash u \Rightarrow* v \Longrightarrow set\ v \subseteq set\ u \cup Syms\ P$
by (*auto simp*: *rtranclp_power dest*!: *deriven_set_subset*)

lemma *derive_Nts_syms_subset*:
$P \vdash u \Rightarrow v \Longrightarrow Nts\_syms\ v \subseteq Nts\_syms\ u \cup Rhs\_Nts\ P$
by(*auto simp*: *Rhs_Nts_def derive_iff*)

lemma *deriven_Nts_syms_subset*:
$P \vdash u \Rightarrow(n) v \Longrightarrow Nts\_syms\ v \subseteq Nts\_syms\ u \cup Rhs\_Nts\ P$
by (*induction n arbitrary*: *u*)
(*auto simp*: *relpowp_Suc_left dest*!: *derive_Nts_syms_subset*)

lemma *derives_Nts_syms_subset*:
$P \vdash u \Rightarrow* v \Longrightarrow Nts\_syms\ v \subseteq Nts\_syms\ u \cup Rhs\_Nts\ P$
by (*auto simp*: *rtranclp_power dest*!: *deriven_Nts_syms_subset*)

lemma *derive_Tms_syms_subset*:
$P \vdash u \Rightarrow v \Longrightarrow Tms\_syms\ v \subseteq Tms\_syms\ u \cup Tms\ P$
by(*auto simp*: *Tms_def derive_iff*)

lemma *deriven_Tms_syms_subset*:
$P \vdash u \Rightarrow(n) v \Longrightarrow Tms\_syms\ v \subseteq Tms\_syms\ u \cup Tms\ P$
by (*induction n arbitrary*: *u*)
(*auto simp*: *relpowp_Suc_left dest*!: *derive_Tms_syms_subset*)

lemma *derives_Tms_syms_subset*:
$P \vdash u \Rightarrow* v \Longrightarrow Tms\_syms\ v \subseteq Tms\_syms\ u \cup Tms\ P$
by (*auto simp*: *rtranclp_power dest*!: *deriven_Tms_syms_subset*)

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* ⊢ *xs* ⇒∗ *ys*
 and *Q xs*
 and ⋀*u A v w*. ⟦ *P* ⊢ *xs* ⇒∗ *u* @ [*Nt A*] @ *v*; *Q* (*u* @ [*Nt A*] @ *v*); (*A*,*w*) ∈ *P* ⟧
⟹ *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* ⊢ *xs* ⇒∗ *ys*
 and *Base*: *Q ys*
 and *Step*: ⋀*u A v w*. ⟦ *P* ⊢ *u* @ [*Nt A*] @ *v* ⇒∗ *ys*; *Q* (*u* @ *w* @ *v*); (*A*,*w*) ∈ *P*
⟧ ⟹ *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*:
 G ⊢ *u* ⇒ *v* ⟹ *G* ⊢ *u*@*w* ⇒ *v*@*w*
apply(*erule derive.cases*)
using *derive.intros* **by** *fastforce*

lemma *derive_prepend*:
 G ⊢ *u* ⇒ *v* ⟹ *G* ⊢ *w*@*u* ⇒ *w*@*v*
apply(*erule derive.cases*)
by (*metis append.assoc derive.intros*)

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

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

lemma *derives_append*:
  $P \vdash u \Rightarrow* v \Longrightarrow P \vdash u@w \Rightarrow* v@w$
  by (*metis deriven_append rtranclp_power*)

lemma *derives_prepend*:
  $P \vdash u \Rightarrow* v \Longrightarrow P \vdash w@u \Rightarrow* w@v$
  by (*metis deriven_prepend rtranclp_power*)

lemma *derive_append_decomp*:
  $P \vdash u@v \Rightarrow w \longleftrightarrow$
  $(\exists\, u'.\ w = u'@v \wedge P \vdash u \Rightarrow u') \vee (\exists\, v'.\ w = u@v' \wedge P \vdash v \Rightarrow v')$
(**is** *?l* $\longleftrightarrow$ *?r*)
proof
  assume *?l*
  then obtain *A r u1 u2*
    where *Ar*: $(A,r) \in P$
      and *uv*: $u@v = u1\ @\ Nt\ A\ \#\ u2$
      and *w*: $w = u1\ @\ r\ @\ u2$
    by (*auto simp*: *derive_iff*)
  from *uv* **have** $(\exists\, s.\ u2 = s\ @\ v \wedge u = u1\ @\ Nt\ A\ \#\ s) \vee$
  $(\exists\, s.\ u1 = u@s \wedge v = s\ @\ Nt\ A\ \#\ u2)$
    by (*auto simp*: *append_eq_append_conv2 append_eq_Cons_conv*)
  with *Ar w* **show** *?r* **by** (*fastforce simp*: *derive_iff*)
next
  show *?r* $\Longrightarrow$ *?l*
    by (*auto simp add*: *derive_append derive_prepend*)
qed

lemma *deriven_append_decomp*:
  $P \vdash u\ @\ v \Rightarrow(n)\ w \longleftrightarrow$
  $(\exists\, n1\ n2\ w1\ w2.\ n = n1 + n2 \wedge w = w1\ @\ w2 \wedge P \vdash u \Rightarrow(n1)\ w1 \wedge P \vdash v$
$\Rightarrow(n2)\ w2)$
  (**is** *?l* $\longleftrightarrow$ *?r*)
proof
  show *?l* $\Longrightarrow$ *?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*: $P \vdash u \Rightarrow u' \wedge v' = v \vee u' = u \wedge P \vdash v \Rightarrow v'$

```
      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 ⊢ [] ⇒ w ⟷ False
```

by (*auto simp*: *derive_iff*)

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

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

lemma *deriven_start1*:
assumes *P* ⊢ [*Nt A*] ⇒(*n*) *map Tm w*
shows ∃ α *m*. *n* = *Suc m* ∧ *P* ⊢ α ⇒(*m*) (*map Tm w*) ∧ (*A*,α) ∈ *P*
proof (*cases n*)
case *0*
thus *?thesis*
using *assms* **by** *auto*
next
case (*Suc m*)
then obtain α **where** ∗: *P* ⊢ [*Nt A*] ⇒ α *P* ⊢ α ⇒(*m*) *map Tm w*
using *assms* **by** (*meson relpowp_Suc_E2*)
from *derive.cases*[*OF* ∗(*1*)] **have** (*A*, α) ∈ *P*
by (*simp add*: *Cons_eq_append_conv*) *blast*
thus *?thesis* **using** ∗(*2*) *Suc* **by** *auto*
qed

lemma *derives_start1*: *P* ⊢ [*Nt A*] ⇒∗ *map Tm w* ⟹ ∃ α. *P* ⊢ α ⇒∗ *map Tm w* ∧ (*A*,α) ∈ *P*
using *deriven_start1* **by** (*metis rtranclp_power*)

lemma *notin_Lhss_iff_Rhss*: *A* ∉ *Lhss P* ⟷ *Rhss P A* = {}
by (*auto simp*: *Lhss_def Rhss_def*)

lemma *Lang_empty_if_notin_Lhss*: *A* ∉ *Lhss P* ⟹ *Lang P A* = {}
unfolding *Lhss_def Lang_def*
using *derives_start1* **by** *fastforce*

lemma *derive_Tm_Cons*:
P ⊢ *Tm a* # *u* ⇒ *v* ⟷ (∃ *w*. *v* = *Tm a* # *w* ∧ *P* ⊢ *u* ⇒ *w*)
by (*fastforce simp*: *derive_iff Cons_eq_append_conv*)

lemma *deriven_Tm_Cons*:
P ⊢ *Tm a* # *u* ⇒(*n*) *v* ⟷ (∃ *w*. *v* = *Tm a* # *w* ∧ *P* ⊢ *u* ⇒(*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 derives_append_append:
  P ⊢ α ⇒* α' ⟹ P ⊢ β ⇒* β' ⟹ P ⊢ α @ β ⇒* α' @ β'
  using derives_append_decomp by blast
```

lemma *derives_append_Nt_Cons*:
$(B,\beta) \in P \Longrightarrow$
$P \vdash \alpha \Rightarrow* \alpha' \Longrightarrow P \vdash \beta \Rightarrow* \beta' \Longrightarrow P \vdash \gamma \Rightarrow* \gamma' \Longrightarrow$
$P \vdash \alpha @ Nt\ B \# \gamma \Rightarrow* \alpha' @ \beta' @ \gamma'$
by (*metis derives_Cons_decomp derives_append_decomp*)

lemma *derives_simul_rules*:
assumes $\bigwedge A\ w.\ (A,w) \in P \Longrightarrow P' \vdash [Nt\ A] \Rightarrow* w$
shows $P \vdash w \Rightarrow* w' \Longrightarrow P' \vdash w \Rightarrow* w'$
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

1.2.4 Derivations leading to terminal words

lemma *derive_decomp_Tm*: $P \vdash \alpha \Rightarrow(n)\ map\ Tm\ \beta \Longrightarrow$
$\exists \beta s\ ns.\ \beta = concat\ \beta s \wedge length\ \alpha = length\ \beta s \wedge length\ \alpha = length\ ns \wedge sum\_list\ ns = n$
$\wedge\ (\forall i < length\ \beta s.\ P \vdash [\alpha\ !\ i] \Rightarrow(ns!i)\ map\ Tm\ (\beta s\ !\ i))$
(**is** $\_ \Longrightarrow \exists \beta s\ ns.\ ?G\ \alpha\ \beta\ n\ \beta s\ ns$)
proof (*induction* α *arbitrary*: $\beta\ n$)
case (*Cons s* α)
from *deriven_Cons_decomp*[*THEN iffD1*, *OF Cons.prems*]
show *?case*
proof (*elim disjE exE conjE*)
fix γ **assume** *as*: $map\ Tm\ \beta = s \# \gamma\ P \vdash \alpha \Rightarrow(n)\ \gamma$
then obtain $s'\ \gamma'$ **where** $\beta = s' \# \gamma'\ \ P \vdash \alpha \Rightarrow(n)\ map\ Tm\ \gamma'\ s = Tm\ s'$ **by** *force*
from *Cons.IH*[*OF this*(*2*)] **obtain** $\beta s\ ns$ **where** $*$: $?G\ \alpha\ \gamma'\ n\ \beta s\ ns$
by *blast*
let $?\beta s = [s']\#\beta s$
let $?ns = 0\#ns$
have $?G\ (s\#\alpha)\ \beta\ n\ ?\beta s\ ?ns$
using $\langle \beta = \_ \rangle$ *as* $*$ **by** (*auto simp*: *nth_Cons'*)
then show *?thesis* **by** *blast*
next
fix *n1 n2 A w* $\beta 1\ \beta 2$
assume $*$: $n = Suc\ (n1 + n2)\ map\ Tm\ \beta = \beta 1 @ \beta 2\ s = Nt\ A\ (A, w) \in P\ P \vdash w \Rightarrow(n1)\ \beta 1\ P \vdash \alpha \Rightarrow(n2)\ \beta 2$
then obtain $\beta 1'\ \beta 2'$ **where** $**$: $\beta = \beta 1' @ \beta 2'\ P \vdash w \Rightarrow(n1)\ map\ Tm\ \beta 1'\ P \vdash \alpha \Rightarrow(n2)\ map\ Tm\ \beta 2'$
by (*metis* (*no_types*, *lifting*) *append_eq_map_conv*)
from *Cons.IH*[*OF this*(*3*)] **obtain** $\beta s\ ns$
where $***$: $?G\ \alpha\ \beta 2'\ n2\ \beta s\ ns$

```
    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: de-
riven_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 de-
riven_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 deriven_Nt_Cons_map_Tm: P ⊢ Nt A # β ⇒(n) map Tm w ⟷
(∃ α m l v u. (A,α) ∈ P ∧ P ⊢ α ⇒(m) map Tm v ∧ P ⊢ β ⇒(l) map Tm u ∧
  n = Suc (m + l) ∧ w = v @ u)
  by (force simp: deriven_Cons_decomp map_eq_append_conv)

lemma deriven_Tm_Cons_map_Tm: P ⊢ Tm a # β ⇒(n) map Tm w ⟷
(∃ v. P ⊢ β ⇒(n) map Tm v ∧ w = a # v)
  by (auto simp: deriven_Tm_Cons)

lemma deriven_Cons_map_Tm:
  P ⊢ x # u ⇒(n) map Tm w ⟷
  (∃ a v2. x = Tm a ∧ w = a # v2 ∧ P ⊢ u ⇒(n) map Tm v2) ∨
  (∃ n1 n2 A α v1 v2. n = Suc (n1 + n2) ∧ w = v1 @ v2 ∧ x = Nt A ∧
   (A,α) ∈ P ∧ P ⊢ α ⇒(n1) map Tm v1 ∧ P ⊢ u ⇒(n2) map Tm v2)
  apply (cases x)
   apply (force simp: deriven_Nt_Cons_map_Tm)
  by (force simp: deriven_Tm_Cons_map_Tm)

lemma deriven_append_map_Tm: P ⊢ α @ β ⇒(n) map Tm w ⟷
  (∃ m l v u. P ⊢ α ⇒(m) map Tm v ∧ P ⊢ β ⇒(l) map Tm u ∧ n = m + l ∧ w
= v @ u)
proof (induction α arbitrary: β n w)
  case Nil
  show ?case by simp
next
  case (Cons x α)
  show ?case
  proof (cases x)
    case x: (Tm a)
    show ?thesis by (force simp: x deriven_Tm_Cons_map_Tm Cons)
  next
    case x: (Nt A)
    show ?thesis
    proof safe
      assume P ⊢ (x # α) @ β ⇒(n) map Tm w
      from this[unfolded x append.simps deriven_Nt_Cons_map_Tm]
      obtain γ m l v u where
        n: n = Suc (m + l) and A: (A,γ) ∈ P and w: w = v @ u
        and γv: P ⊢ γ ⇒(m) map Tm v and αβ: P ⊢ α @ β ⇒(l) map Tm u
        by auto
```

```
      from A γv have Av: P ⊢ [Nt A] ⇒(Suc m) map Tm v
        by (simp add: derive_singleton relpowp_Suc_I2)
      from αβ[unfolded Cons]
      obtain k j t s where l: l = k + j and u: u = t @ s
        and α: P ⊢ α ⇒(k) map Tm t and β: P ⊢ β ⇒(j) map Tm s by auto
      from Av α have xα: P ⊢ x # α ⇒(Suc m + k) map Tm (v @ t)
        by (force simp: x deriven_Nt_Cons_map_Tm simp del: map_append)
      show ∃ m l v u.
       P ⊢ x # α ⇒(m) map Tm v ∧
       P ⊢ β ⇒(l) map Tm u ∧ n = m + l ∧
       w = v @ u
        apply (intro exI conjI)
           apply (fact xα)
          apply (fact β)
        by (auto simp: n l w u)
    next
      fix m l v u
      assume n: n = m + l and w: w = v @ u
        and xα: P ⊢ x # α ⇒(m) map Tm v
        and β: P ⊢ β ⇒(l) map Tm u
      from xα[unfolded x deriven_Nt_Cons_map_Tm]
      obtain γ k j t s where
        m: m = Suc (k + j) and A: (A,γ) ∈ P and v: v = t @ s
        and γ: P ⊢ γ ⇒(k) map Tm t and α: P ⊢ α ⇒(j) map Tm s
        by auto
      show P ⊢ (x # α) @ β ⇒(m + l) map Tm (v @ u)
        apply (unfold x append.simps deriven_Nt_Cons_map_Tm)
      proof (intro exI conjI)
        show m + l = Suc (k + (j + l)) by (simp add: m)
        show (A,γ) ∈ P using A.
        show v @ u = t @ s @ u by (simp add: v)
        show  P ⊢ γ ⇒(k) map Tm t using γ.
        from α β show αβ: P ⊢ α @ β ⇒(j+l) map Tm (s @ u)
          by (unfold Cons, auto)
      qed
    qed
  qed
qed

lemma deriven_Nt_map_Tm: P ⊢ α @ Nt B # γ ⇒(n) map Tm w ⟷
(∃ β m l k v u t. (B,β) ∈ P ∧
  P ⊢ α ⇒(m) map Tm v ∧ P ⊢ β ⇒(l) map Tm u ∧ P ⊢ γ ⇒(k) map Tm t ∧
  n = Suc (m + l + k) ∧ w = v @ u @ t)
  by (force simp: deriven_append_map_Tm deriven_Nt_Cons_map_Tm)

lemma map_Tm_Nt_eq_map_Tm_Nt:
  map Tm xs @ Nt y # zs = map Tm xs' @ Nt y' # zs' ⟷ xs = xs' ∧ y = y' ∧
zs = zs'
  apply (subst append_Cons_eq_append_Cons)
```

by *auto*

lemma *deriven_Suc_map_Tm_decomp*: $P \vdash \alpha \Rightarrow(Suc\ n)\ map\ Tm\ w \longleftrightarrow$
$(\exists\, v\ B\ \beta\ \gamma\ u\ t\ m\ l.\ (B,\beta) \in P \wedge P \vdash \beta \Rightarrow(m)\ map\ Tm\ u \wedge P \vdash \gamma \Rightarrow(l)\ map\ Tm\ t \wedge$
$n = m + l \wedge \alpha = map\ Tm\ v\ @\ Nt\ B\ \#\ \gamma \wedge w = v\ @\ u\ @\ t)$
by (*fastforce simp*: *deriven_Suc_decomp_left map_eq_append_conv map_Tm_Nt_eq_map_Tm_Nt append_eq_map_conv*)

lemma *derives_append_map_Tm*:
$P \vdash \alpha\ @\ \beta \Rightarrow* map\ Tm\ w \longleftrightarrow$
$(\exists\, v\ u.\ P \vdash \alpha \Rightarrow* map\ Tm\ v \wedge P \vdash \beta \Rightarrow* map\ Tm\ u \wedge w = v\ @\ u)$
by (*force simp*: *rtranclp_power deriven_append_map_Tm*)

lemma *derives_Nt_map_Tm*:
$P \vdash \alpha\ @\ Nt\ B\ \#\ \gamma \Rightarrow* map\ Tm\ w \longleftrightarrow$
$(\exists\, \beta\ v\ u\ t.\ (B,\beta) \in P \wedge$
$P \vdash \alpha \Rightarrow* map\ Tm\ v \wedge P \vdash \beta \Rightarrow* map\ Tm\ u \wedge P \vdash \gamma \Rightarrow* map\ Tm\ t \wedge$
$w = v\ @\ u\ @\ t)$
by (*force simp*: *rtranclp_power deriven_Nt_map_Tm*)

lemma *derives_Nt_Cons_map_Tm*:
$P \vdash Nt\ A\ \#\ \beta \Rightarrow* map\ Tm\ w \longleftrightarrow$
$(\exists\, \alpha\ v\ u.\ (A,\alpha) \in P \wedge P \vdash \alpha \Rightarrow* map\ Tm\ v \wedge P \vdash \beta \Rightarrow* map\ Tm\ u \wedge w = v\ @\ u)$
using *derives_Nt_map_Tm*[**where** $\alpha = []$] **by** *simp*

lemma *derives_Nt_Cons_Lang*:
$P \vdash Nt\ A\ \#\ \alpha \Rightarrow* map\ Tm\ w \longleftrightarrow (\exists\, v\ u.\ v \in Lang\ P\ A \wedge P \vdash \alpha \Rightarrow* map\ Tm\ u \wedge w = v\ @\ u)$
by (*force simp*: *derives_Cons_decomp Lang_def map_eq_Cons_conv map_eq_append_conv*)

lemma *Lang_of_Nil*[*simp*]: *Lang_of P* $[] = \{[]\}$
by (*auto simp*: *Lang_of_def*)

lemma *Lang_of_iff_derives*: $w \in Lang\_of\ P\ \alpha \longleftrightarrow P \vdash \alpha \Rightarrow* map\ Tm\ w$
by (*auto simp*: *Lang_of_def*)

lemma *Lang_ofE_deriven*:
assumes $w \in Lang\_of\ P\ \alpha$ **and** $\bigwedge n.\ P \vdash \alpha \Rightarrow(n)\ map\ Tm\ w \Longrightarrow thesis$
shows *thesis*
using *assms* **by** (*auto simp*: *Lang_of_iff_derives rtranclp_power*)

lemma *Lang_of_Tm_Cons*: *Lang_of P* $(Tm\ a\ \#\ \alpha) = \{[a]\}$ *@@ Lang_of P* α
by (*auto simp*: *Lang_of_def derives_Tm_Cons conc_def*)

lemma *Lang_of_map_Tm*: *Lang_of P* $(map\ Tm\ w) = \{w\}$
by (*induction w*, *simp_all add*: *Lang_of_Tm_Cons insert_conc*)

lemma *Lang_of_Nt_Cons*: *Lang_of P* (*Nt A # α*) = *Lang P A @@ Lang_of P α*
by (*force simp add*: *Lang_of_def Lang_def derives_Cons_decomp map_eq_append_conv conc_def*)

lemma *Lang_of_Cons*: *Lang_of P* (*x # α*) = (*case x of Tm a ⇒ {*[*a*]*} | Nt A ⇒ Lang P A*) *@@ Lang_of P α*
by (*simp add*: *Lang_of_Tm_Cons Lang_of_Nt_Cons split*: *sym.splits*)

lemma *Lang_of_append*: *Lang_of P* (*α @ β*) = *Lang_of P α @@ Lang_of P β*
by (*induction α arbitrary*: *β*, *simp_all add*: *Lang_of_Cons conc_assoc split*: *sym.splits*)

lemma *Lang_of_set_conc*: *Lang_of_set P* (*X @@ Y*) = *Lang_of_set P X @@ Lang_of_set P Y*
by (*force simp*: *Lang_of_append elim*!: *concE*)

lemma *Lang_of_set_Rhss*: *Lang_of_set P* (*Rhss P A*) = *Lang P A*
by (*auto simp*: *Lang_def Lang_of_def Rhss_def converse_rtranclp_into_rtranclp derive_singleton*
dest: *derives_start1*)

lemma *Lang_of_prod_subset*: (*A*,*α*) ∈ *P* ⟹ *Lang_of P α* ⊆ *Lang P A*
apply (*fold Lang_of_set_Rhss*) **by** (*auto simp*: *Rhss_def*)

lemma *Lang_le_iff_Lang_of_le*: *Lang P* ≤ *Lang P′* ⟷ *Lang_of P* ≤ *Lang_of P′*
proof (*safe intro*!: *le_funI*)
fix *α w*
assume *le*: *Lang P* ≤ *Lang P′* **and** *w*: *w* ∈ *Lang_of P α*
from *w* **show** *w* ∈ *Lang_of P′ α*
apply (*induction α arbitrary*: *w*)
using *le*[*THEN le_funD*, *THEN subsetD*]
by (*auto simp*: *Lang_of_Cons insert_conc split*: *sym.splits*)
next
fix *A w*
assume *le*: *Lang_of P* ≤ *Lang_of P′* **and** *w*: *w* ∈ *Lang P A*
from *le*[*THEN le_funD*, *of* [*Nt A*]] *w*
show *w* ∈ *Lang P′ A* **by** (*auto simp*: *Lang_of_Cons*)
qed

lemma *Lang_eq_iff_Lang_of_eq*: *Lang P* = *Lang P′* ⟷ *Lang_of P* = *Lang_of P′*
apply (*subst eq_iff*) **by** (*auto simp*: *Lang_le_iff_Lang_of_le*)

lemma *Lang_of_le_iff_derives*:
Lang_of P ≤ *Lang_of P′* ⟷ (∀ *α w*. *P* ⊢ *α* ⇒∗ *map Tm w* ⟶ *P′* ⊢ *α* ⇒∗ *map Tm w*)
by (*auto simp*: *Lang_of_def le_fun_def*)

lemma *Lang_le_iff_derives*:
Lang P $\leq$ *Lang P'* $\longleftrightarrow$ ($\forall \alpha\ w.\ P \vdash \alpha \Rightarrow* map\ Tm\ w \longrightarrow P' \vdash \alpha \Rightarrow* map\ Tm\ w$)
by (*simp only*: *Lang_le_iff_Lang_of_le Lang_of_le_iff_derives*)

lemma *Lang_eq_iff_derives*:
Lang P = *Lang P'* $\longleftrightarrow$ ($\forall \alpha\ w.\ P \vdash \alpha \Rightarrow* map\ Tm\ w \longleftrightarrow P' \vdash \alpha \Rightarrow* map\ Tm\ w$)
apply (*subst eq_iff*) **by** (*auto simp*: *Lang_le_iff_derives*)

lemma *Rhss_le_Ders*: *Rhss P* $\leq$ *Ders P*
by (*auto simp*: *le_fun_def Rhss_def Ders_def derive_singleton*)

lemma *Lang_of_set_pow*: *Lang_of_set P* (*X* $^\frown$ *n*) = *Lang_of_set P X* $^\frown$ *n*
by (*induction n*, *simp_all add*: *Lang_of_set_conc*)

lemma *Lang_of_set_star*: *Lang_of_set P* (*star X*) = *star* (*Lang_of_set P X*)
by (*auto simp*: *star_def Lang_of_set_pow*)

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* $\Rightarrow$ ('*n*,'*t*) *syms* $\Rightarrow$ ('*n*,'*t*) *syms* $\Rightarrow$ *bool*
((*2*_ $\vdash$/ (_/ $\Rightarrow$*bu*/ _)) [*50*, *0*, *50*] *50*) **for** *P* :: ('*n*, '*t*) *Prods*
where
bu_refl: *P* $\vdash \alpha \Rightarrow bu\ \alpha$ |
bu_prod: (*A*,α) $\in$ *P* $\Longrightarrow$ *P* $\vdash$ [*Nt A*] $\Rightarrow bu\ \alpha$ |
bu_embed: $[\![$ *P* $\vdash \alpha \Rightarrow bu\ \alpha_1$ @ α_2 @ α_3; *P* $\vdash \alpha_2 \Rightarrow bu\ \beta\ ]\!] \Longrightarrow$ *P* $\vdash \alpha \Rightarrow bu\ \alpha_1$ @ β @ α_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 α_1 α_2 α_3 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*)

1.2.5 Leftmost/Rightmost Derivations

inductive *derivel* :: ('*n*,'*t*) *Prods* ⇒ ('*n*,'*t*) *syms* ⇒ ('*n*,'*t*)*syms* ⇒ *bool*
(($2\_ \vdash / (\_ \Rightarrow l / \_)$) [*50*, *0*, *50*] *50*) **where**
$(A,\alpha) \in P \Longrightarrow P \vdash map\ Tm\ u\ @\ Nt\ A\ \#\ v \Rightarrow l\ map\ Tm\ u\ @\ \alpha\ @\ v$

abbreviation *derivels* (($2\_ \vdash / (\_ \Rightarrow l* / \_)$) [*50*, *0*, *50*] *50*) **where**
$P \vdash u \Rightarrow l*\ v \equiv ((derivel\ P)\ \widehat{}**)\ u\ v$

abbreviation *derivels1* (($2\_ \vdash / (\_ \Rightarrow l+ / \_)$) [*50*, *0*, *50*] *50*) **where**
$P \vdash u \Rightarrow l+\ v \equiv ((derivel\ P)\ \widehat{}++)\ u\ v$

abbreviation *deriveln* (($2\_ \vdash / (\_ \Rightarrow l'(\_') / \_)$) [*50*, *0*, *0*, *50*] *50*) **where**
$P \vdash u \Rightarrow l(n)\ v \equiv ((derivel\ P)\ \widehat{\widehat{}}\ n)\ u\ v$

inductive *deriver* :: ('*n*,'*t*) *Prods* ⇒ ('*n*,'*t*) *syms* ⇒ ('*n*,'*t*)*syms* ⇒ *bool*
(($2\_ \vdash / (\_ \Rightarrow r / \_)$) [*50*, *0*, *50*] *50*) **where**
$(A,\alpha) \in P \Longrightarrow P \vdash u\ @\ Nt\ A\ \#\ map\ Tm\ v \Rightarrow r\ u\ @\ \alpha\ @\ map\ Tm\ v$

abbreviation *derivers* (($2\_ \vdash / (\_ \Rightarrow r* / \_)$) [*50*, *0*, *50*] *50*) **where**
$P \vdash u \Rightarrow r*\ v \equiv ((deriver\ P)\ \widehat{}**)\ u\ v$

abbreviation *derivers1* (($2\_ \vdash / (\_ \Rightarrow r+ / \_)$) [*50*, *0*, *50*] *50*) **where**
$P \vdash u \Rightarrow r+\ v \equiv ((deriver\ P)\ \widehat{}++)\ u\ v$

abbreviation *derivern* (($2\_ \vdash / (\_ \Rightarrow r'(\_') / \_)$) [*50*, *0*, *0*, *50*] *50*) **where**
$P \vdash u \Rightarrow r(n)\ v \equiv ((deriver\ P)\ \widehat{\widehat{}}\ n)\ u\ v$

lemma *derivel_iff*: $R \vdash u \Rightarrow l\ v \longleftrightarrow$
$(\exists\ (A,w) \in R.\ \exists\, u1\ u2.\ u = map\ Tm\ u1\ @\ Nt\ A\ \#\ u2 \wedge v = map\ Tm\ u1\ @\ w\ @\ u2)$
by (*auto simp*: *derivel.simps*)

lemma *derivel_from_empty*[*simp*]:
$P \vdash [] \Rightarrow l\ w \longleftrightarrow False$ **by** (*auto simp*: *derivel_iff*)

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

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

lemma *Un_derivel*: $R \cup S \vdash y \Rightarrow l\ z \longleftrightarrow R \vdash y \Rightarrow l\ z \vee S \vdash y \Rightarrow l\ z$
by (*fastforce simp*: *derivel_iff*)

lemma *derivel_append*:
$P \vdash u \Rightarrow l\ v \Longrightarrow P \vdash u\ @\ w \Rightarrow l\ v\ @\ w$
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 (*fastforce 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_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 ⊢ *map Tm w* @ *u* ⇒*l*(*n*) *v* ⟷ (∃ *x*. *v* = *map Tm w* @ *x* ∧ *P* ⊢ *u* ⇒*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 ⊢ *map Tm w* @ *u* ⇒*l*∗ *v* ⟷ (∃ *x*. *v* = *map Tm w* @ *x* ∧ *P* ⊢ *u* ⇒*l*∗ *x*)
by (*metis deriveln_map_Tm_append rtranclp_power*)

lemma *derivel_Nt_Cons*:
P ⊢ *Nt A* # *u* ⇒*l v* ⟷ (∃ *w*. (*A*,*w*) ∈ *P* ∧ *v* = *w* @ *u*)
by (*auto simp*: *derivel_iff Cons_eq_append_conv Cons_eq_map_conv*)

lemma *derivels1_Nt_Cons*:
P ⊢ *Nt A* # *u* ⇒*l*+ *v* ⟷ (∃ *w*. (*A*,*w*) ∈ *P* ∧ *P* ⊢ *w* @ *u* ⇒*l*∗ *v*)
by (*auto simp*: *tranclp_unfold_left derivel_Nt_Cons OO_def*)

lemma *derivels_Nt_Cons*:
P ⊢ *Nt A* # *u* ⇒*l*∗ *v* ⟷ *v* = *Nt A* # *u* ∨ (∃ *w*. (*A*,*w*) ∈ *P* ∧ *P* ⊢ *w* @ *u* ⇒*l*∗ *v*)
by (*auto simp*: *Nitpick.rtranclp_unfold derivels1_Nt_Cons*)

lemma *deriveln_Nt_Cons*:
P ⊢ *Nt A* # *u* ⇒*l*(*n*) *v* ⟷ (
case n of 0 ⇒ *v* = *Nt A* # *u*
| *Suc m* ⇒ ∃ *w*. (*A*,*w*) ∈ *P* ∧ *P* ⊢ *w* @ *u* ⇒*l*(*m*) *v*)
by (*cases n*) (*auto simp*: *derivel_Nt_Cons relpowp_Suc_left OO_def*)

lemma *derivel_Cons*:
P ⊢ *x* # *u* ⇒*l v* ⟷
(*case x of Nt A* ⇒ ∃ *w*. (*A*,*w*) ∈ *P* ∧ *v* = *w* @ *u* | *Tm a* ⇒ ∃ *w*. *v* = *Tm a* # *w* ∧ *P* ⊢ *u* ⇒*l w*)
by (*auto simp*: *derivel_Nt_Cons derivel_Tm_Cons split*: *sym.splits*)

lemma *deriveln_Cons*:
P ⊢ *x* # *u* ⇒*l*(*n*) *v* ⟷ (
case n of 0 ⇒ *v* = *x* # *u*
| *Suc m* ⇒ (
case x of Nt A ⇒ ∃ *w*. (*A*,*w*) ∈ *P* ∧ *P* ⊢ *w* @ *u* ⇒*l*(*m*) *v*
| *Tm a* ⇒ ∃ *w*. *v* = *Tm a* # *w* ∧ *P* ⊢ *u* ⇒*l*(*n*) *w*))
by (*auto simp*: *deriveln_Nt_Cons deriveln_Tm_Cons split*: *nat.splits sym.splits*)

lemma *derivel_not_elim_Tm*:
P ⊢ *xs* ⇒*l map Nt w* ⟹ ∃ *v*. *xs* = *map Nt v*
by (*cases xs*)
(*auto simp*: *derivel_Cons Cons_eq_map_conv map_eq_append_conv split*: *sym.splits*)

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 *derivel_append_iff*:
$P \vdash u@v \Rightarrow l\ w \longleftrightarrow$
$(\exists u'.\ w = u'@v \wedge P \vdash u \Rightarrow l\ u') \vee (\exists u'\ v'.\ w = u@v' \wedge u = map\ Tm\ u' \wedge P \vdash v \Rightarrow l\ v')$
(**is** *?l* $\longleftrightarrow$ *?r*)
proof
assume *?l*
then obtain *A r u1 u2*
where *Ar*: $(A,r) \in P$
and *uv*: $u@v = map\ Tm\ u1\ @\ Nt\ A\ \#\ u2$
and *w*: $w = map\ Tm\ u1\ @\ r\ @\ u2$
by (*auto simp*: *derivel_iff*)
from *uv* **have** *case_dist*: $(\exists s.\ u2 = s\ @\ v \wedge u = map\ Tm\ u1\ @\ Nt\ A\ \#\ s) \vee (\exists s.\ map\ Tm\ u1 = u\ @\ s\ \wedge v = s\ @\ Nt\ A\ \#\ u2)$ (**is** *?h1* $\vee$ *?h2*)
by (*auto simp*: *append_eq_append_conv2 append_eq_Cons_conv*)
show *?r* **proof** (*rule disjE*[*OF case_dist*])
assume *?h1*
with *Ar w* **show** *?thesis* **by** (*fastforce simp*: *derivel_iff*)
next
assume *?h2*
then obtain *s* **where** *map_u1_def*: $map\ Tm\ u1 = u\ @\ s$ **and** *v_def*: $v = s\ @\ Nt\ A\ \#\ u2$ **by** *blast*

from *map_u1_def* **obtain** u' s' **where** *u_def*: $u = map\ Tm\ u'$ **and** *s_def*: $s = map\ Tm\ s'$
using *append_eq_map_conv*[*of u s Tm u1*] **by** *auto*

from *w map_u1_def s_def* **have** $w = u\ @\ (map\ Tm\ s'\ @\ r\ @\ u2)$ **by** *simp*

moreover from *Ar v_def s_def* **have** $P \vdash v \Rightarrow l\ map\ Tm\ s'\ @\ r\ @\ u2$
using *derivel_iff*[*of P*] **by** *blast*

ultimately show *?thesis*
using *u_def* **by** *blast*
qed
next
show *?r* $\Longrightarrow$ *?l*
by (*auto simp add*: *derivel_append derivel_map_Tm_append*)
qed

lemma *deriveln_ConsD*:
assumes $P \vdash x\#v \Rightarrow l(n)\ u$
shows $(\exists\, u'.\ u = u'\ @\ v \wedge P \vdash [x] \Rightarrow l(n)\ u') \vee (\exists\, w_1\ u_2\ m_1\ m_2.\ m_1 + m_2 = n \wedge u = map\ Tm\ w_1\ @\ u_2$
$\wedge\ P \vdash [x] \Rightarrow l(m_1)\ map\ Tm\ w_1 \wedge P \vdash v \Rightarrow l(m_2)\ u_2)$
using *assms* **proof** (*induction n arbitrary*: *u*)
case (*Suc n*)
from *Suc(2)* **obtain** w **where** *x_v_deriveln_w*: $P \vdash x\ \#\ v \Rightarrow l(n)\ w$ **and** *w_derivel_u*: $P \vdash w \Rightarrow l\ u$
by (*metis relpowp_Suc_E*)
from *Suc(1)*[*OF x_v_deriveln_w*] **have** *IH*: $(\exists\, u'.\ w = u'\ @\ v \wedge P \vdash [x] \Rightarrow l(n)\ u') \vee$
$(\exists\, w_1\ u_2\ m_1\ m_2.\ m_1 + m_2 = n \wedge w = map\ Tm\ w_1\ @\ u_2 \wedge P \vdash [x] \Rightarrow l(m_1)\ map\ Tm\ w_1 \wedge P \vdash v \Rightarrow l(m_2)\ u_2)$ (**is** *?l* $\vee$ *?r*) **.**
show *?case* **proof** (*rule disjE*[*OF IH*])
assume *?l*
then obtain u' **where** *w_def*: $w = u'\ @\ v$ **and** *x_deriveln_u'*: $P \vdash [x] \Rightarrow l(n)\ u'$ **by** *blast*
from *w_def w_derivel_u* **have** $P \vdash u'\ @\ v \Rightarrow l\ u$ **by** *simp*
hence *case_dist*: $(\exists\, u_0.\ u = u_0\ @\ v \wedge P \vdash u' \Rightarrow l\ u_0) \vee$
$(\exists\, u_1\ u_2.\ u = u'\ @\ u_2 \wedge u' = map\ Tm\ u_1 \wedge P \vdash v \Rightarrow l\ u_2)$ (**is** *?h1* $\vee$ *?h2*)
using *derivel_append_iff*[*of P u' v u*] **by** *simp*
show *?thesis* **proof** (*rule disjE*[*OF case_dist*])
assume *?h1*
then obtain u_0 **where** *u_def*: $u = u_0\ @\ v$ **and** *u'_derivel_u0*: $P \vdash u' \Rightarrow l\ u_0$ **by** *blast*
from *x_deriveln_u' u'_derivel_u0* **have** $P \vdash [x] \Rightarrow l(Suc\ n)\ u_0$ **by** (*simp add*: *relpowp_Suc_I*)
with *u_def* **show** *?thesis* **by** *blast*
next

assume *?h2*
then obtain u_1 u_2 **where** *u_def*: $u = u' @ u_2$ **and** *u'_def*: $u' = map\ Tm\ u_1$ **and** *v_derivel_u2*: $P \vdash v \Rightarrow l\ u_2$ **by** *blast*
from *x_deriveln_u' u'_def* **have** $P \vdash [x] \Rightarrow l(n)\ map\ Tm\ u_1$ **by** *simp*
with *u_def u'_def v_derivel_u2* **show** *?thesis* **by** *fastforce*
qed
next
assume *?r*
then obtain w_1 u_2 m_1 m_2 **where** *m1_m2_n*: $m_1 + m_2 = n$ **and** *w_def*: $w = map\ Tm\ w_1 @ u_2$ **and**
x_derivelm1_w1: $P \vdash [x] \Rightarrow l(m_1)\ map\ Tm\ w_1$ **and** *v_derivelm2_u2*: $P \vdash v \Rightarrow l(m_2)\ u_2$ **by** *blast*
from *w_def w_derivel_u* **have** $P \vdash map\ Tm\ w_1 @ u_2 \Rightarrow l\ u$ **by** *simp*
then obtain u' **where** *u_def*: $u = map\ Tm\ w_1 @ u'$ **and** *u2_derivel_u'*: $P \vdash u_2 \Rightarrow l\ u'$
using *derivel_map_Tm_append* **by** *blast*

from *m1_m2_n* **have** $m_1 + Suc\ m_2 = Suc\ n$ **by** *simp*

moreover from *v_derivelm2_u2 u2_derivel_u'* **have** $P \vdash v \Rightarrow l(Suc\ m_2)\ u'$
by (*simp add*: *relpowp_Suc_I*)

ultimately show *?thesis*
using *u_def x_derivelm1_w1* **by** *blast*
qed
qed *simp*

lemma *deriveln_Cons_TmsD*:
assumes $P \vdash x\#v \Rightarrow l(n)\ map\ Tm\ w$
shows $\exists\, w_1\ w_2\ m_1\ m_2.\ m_1 + m_2 = n \wedge w = w_1 @ w_2 \wedge P \vdash [x] \Rightarrow l(m_1)\ map\ Tm\ w_1 \wedge P \vdash v \Rightarrow l(m_2)\ map\ Tm\ w_2$
proof −
have *case_dist*: $(\exists\, u'.\ map\ Tm\ w = u' @ v \wedge P \vdash [x] \Rightarrow l(n)\ u') \vee (\exists\, w_1\ u_2\ m_1\ m_2.\ m_1 + m_2 = n \wedge map\ Tm\ w = map\ Tm\ w_1 @ u_2 \wedge P \vdash [x] \Rightarrow l(m_1)\ map\ Tm\ w_1 \wedge P \vdash v \Rightarrow l(m_2)\ u_2)$ (**is** *?l* ∨ *?r*)
using *deriveln_ConsD*[*OF assms*] **by** *simp*
show *?thesis* **proof** (*rule disjE*[*OF case_dist*])
assume *?l*
then obtain u' **where** *map_w_def*: $map\ Tm\ w = u' @ v$ **and** *x_derives_u'*: $P \vdash [x] \Rightarrow l(n)\ u'$ **by** *blast*
from *map_w_def* **obtain** w_1 w_2 **where** $w = w_1 @ w_2$ **and** *map_w1_def*: $map\ Tm\ w_1 = u'$ **and** $map\ Tm\ w_2 = v$
using *map_eq_append_conv*[*of Tm w u' v*] **by** *blast*

moreover from *x_derives_u' map_w1_def* **have** $P \vdash [x] \Rightarrow l(n)\ map\ Tm\ w_1$ **by** *simp*

moreover have $P \vdash map\ Tm\ w_2 \Rightarrow l(0)\ map\ Tm\ w_2$ **by** *simp*

```
    ultimately show ?thesis by force
  next
    assume ?r
   then obtain w₁ u₂ m₁ m₂ where m1_m2_n: m₁ + m₂ = n and map_w_def:
map Tm w = map Tm w₁ @ u₂
                                        and x_derivelm1_w1: P ⊢ [x] ⇒l(m₁) map
Tm w₁ and v_derivelm2_u2: P ⊢ v ⇒l(m₂) u₂ by blast
    from map_w_def obtain w₁' u₂' where w = w₁' @ u₂' and map (Tm) w₁
= map Tm w₁' and u₂ = map (Tm) u₂'
      using map_eq_append_conv[of Tm w map Tm w₁ u₂] by auto
    with m1_m2_n x_derivelm1_w1 v_derivelm2_u2 show ?thesis by auto
  qed
qed

lemma deriveln_imp_deriven:
  P ⊢ u ⇒l(n) v ⟹ P ⊢ u ⇒(n) v
  using relpowp_mono derivel_imp_derive by metis

lemma derivels_imp_derives:
  P ⊢ u ⇒l* v ⟹ P ⊢ u ⇒* v
  by (metis derivel_imp_derive mono_rtranclp)

lemma deriveln_iff_deriven:
  P ⊢ u ⇒l(n) map Tm v ⟷ P ⊢ u ⇒(n) map Tm v
  (is ?l ⟷ ?r)
proof
  show ?l ⟹ ?r using deriveln_imp_deriven.
  assume ?r
  then show ?l
  proof (induction n arbitrary: u v rule: less_induct)
    case (less n)
    from ‹P ⊢ u ⇒(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*: $R \vdash u \Rightarrow r\ v \longleftrightarrow$
(∃ (*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*: $R \vdash u \Rightarrow r\ v \Longrightarrow R \vdash u \Rightarrow v$
by (*auto simp*: *deriver_iff derive_iff*)

lemma *derivern_imp_deriven*: $R \vdash u \Rightarrow r(n)\ v \Longrightarrow R \vdash u \Rightarrow (n)\ v$
by (*metis* (*no_types*, *lifting*) *deriver_imp_derive relpowp_mono*)

lemma *derivers_imp_derives*: $R \vdash u \Rightarrow r*\ v \Longrightarrow R \vdash u \Rightarrow *\ v$
by (*metis deriver_imp_derive mono_rtranclp*)

lemma *deriver_iff_rev_derivel*:
$P \vdash u \Rightarrow r\ v \longleftrightarrow$ *map_prod id rev* ‘ $P \vdash$ *rev u* $\Rightarrow l$ *rev v* (**is** *?l* $\longleftrightarrow$ *?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 ‘ $P \vdash u \Rightarrow r\ v \longleftrightarrow P \vdash$ *rev u* $\Rightarrow l$ *rev v*
by (*simp add*: *deriver_iff_rev_derivel image_image prod.map_comp o_def*)

lemma *derivern_iff_rev_deriveln*:
$P \vdash u \Rightarrow r(n)\ v \longleftrightarrow$ *map_prod id rev* ‘ $P \vdash$ *rev u* $\Rightarrow l(n)$ *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*:

$P \vdash u$ @ $map\ Tm\ w \Rightarrow r(n)\ v \longleftrightarrow (\exists x.\ v = x$ @ $map\ Tm\ w \wedge P \vdash u \Rightarrow r(n)\ x)$
by (*fastforce simp*: *derivern_iff_rev_deriveln rev_map deriveln_map_Tm_append rev_eq_append_conv*)

lemma *deriver_snoc_Nt*:
$P \vdash u$ @ $[Nt\ A] \Rightarrow r\ v \longleftrightarrow (\exists w.\ (A,w) \in P \wedge v = u$ @ $w)$
by (*force simp*: *deriver_iff_rev_derivel derivel_Nt_Cons rev_eq_append_conv*)

lemma *deriver_singleton*:
$P \vdash [Nt\ A] \Rightarrow r\ v \longleftrightarrow (A,v) \in P$
using *deriver_snoc_Nt*[*of P* []] **by** *auto*

lemma *derivers1_snoc_Nt*:
$P \vdash u$ @ $[Nt\ A] \Rightarrow r{+}\ v \longleftrightarrow (\exists w.\ (A,w) \in P \wedge P \vdash u$ @ $w \Rightarrow r{*}\ v)$
by (*auto simp*: *tranclp_unfold_left deriver_snoc_Nt OO_def*)

lemma *derivers_snoc_Nt*:
$P \vdash u$ @ $[Nt\ A] \Rightarrow r{*}\ v \longleftrightarrow v = u$ @ $[Nt\ A] \vee (\exists w.\ (A,w) \in P \wedge P \vdash u$ @ $w \Rightarrow r{*}\ v)$
by (*auto simp*: *Nitpick.rtranclp_unfold derivers1_snoc_Nt*)

lemma *derivern_snoc_Tm*:
$P \vdash u$ @ $[Tm\ a] \Rightarrow r(n)\ v \longleftrightarrow (\exists w.\ v = w$ @ $[Tm\ a] \wedge P \vdash u \Rightarrow r(n)\ w)$
by (*force simp*: *derivern_iff_rev_deriveln deriveln_Tm_Cons*)

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

lemma *derivern_singleton*:
$P \vdash [Nt\ A] \Rightarrow r(n)\ v \longleftrightarrow$ (
case n of 0 $\Rightarrow v = [Nt\ A]$
| *Suc m* $\Rightarrow \exists w.\ (A,w) \in P \wedge P \vdash w \Rightarrow r(m)\ v$)
using *derivern_snoc_Nt*[*of n P* [] *A v*] **by** (*cases n*, *auto*)

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

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

1.3 Redundant Productions

Productions of the form $A \to A$ are redundant.

lemma *no_self_loops_derive*:
$reflclp\ (derive\ \{(A,\alpha) \in P.\ \alpha \neq [Nt\ A]\}) = reflclp\ (derive\ P)$
by (*force simp*: *fun_eq_iff derive_iff*)

lemma *no_self_loops_derives*:
$\{(A,\alpha) \in P.\ \alpha \neq [Nt\ A]\} \vdash u \Rightarrow* v \longleftrightarrow P \vdash u \Rightarrow* v$
apply (*subst rtranclp_reflclp*[*symmetric*])
by (*simp add*: *no_self_loops_derive*)

lemma *Lang_of_no_self_loops*:
$Lang\_of\ \{(A,\alpha) \in P.\ \alpha \neq [Nt\ A]\} = Lang\_of\ P$
by (*simp add*: *fun_eq_iff Lang_of_def no_self_loops_derives*)

lemma *Lang_no_self_loops*:
$Lang\ \{(A,\alpha) \in P.\ \alpha \neq [Nt\ A]\} = Lang\ P$
by (*simp add*: *Lang_eq_iff_Lang_of_eq Lang_of_no_self_loops*)

lemma *Lang_eq_Rhss_no_self_loop*:
$Lang\ P\ A = Lang\_of\_set\ P\ (Rhss\ P\ A - \{[Nt\ A]\})$
proof−
have $Lang\ P\ A = Lang\ \{(A,\alpha) \in P.\ \alpha \neq [Nt\ A]\}\ A$
by (*simp add*: *Lang_no_self_loops*)
also have $\ldots = Lang\_of\_set\ \{(A,\alpha) \in P.\ \alpha \neq [Nt\ A]\}\ (Rhss\ P\ A - \{[Nt\ A]\})$
by (*auto simp*: *Lang_of_set_Rhss*[*symmetric*] *Rhss_def*)
finally show *?thesis* **by** (*simp add*: *Lang_of_no_self_loops*)
qed

lemma *no_self_loops_derivel*:

```
  reflclp (derivel {(A,α) ∈ P. α ≠ [Nt A]}) = reflclp (derivel P)
  by (force simp: fun_eq_iff derivel_iff)

lemma no_self_loops_derivels:
  {(A,α) ∈ P. α ≠ [Nt A]} ⊢ u ⇒l* v ⟷ P ⊢ u ⇒l* v
  apply (subst rtranclp_reflclp[symmetric])
  by (simp add: no_self_loops_derivel)
```

Rules that can be simulated by other rules are redundant.

```
lemma Rhss_le_Ders_imp_Lang_le: assumes Rhss P ≤ Ders P' shows Lang
P ≤ Lang P'
  apply (unfold Lang_le_iff_derives)
proof (intro allI impI)
  fix α w
  assume P ⊢ α ⇒* map Tm w
  then obtain n where P ⊢ α ⇒(n) map Tm w by (auto simp: rtranclp_power)
  then show P' ⊢ α ⇒* map Tm w
  proof (induction n arbitrary: α w rule: less_induct)
    case (less n')
    show ?case
    proof (cases n')
      case 0
      with less.prems show ?thesis by simp
    next
      case [simp]: (Suc n)
      from less.prems[unfolded this deriven_Suc_map_Tm_decomp]
      obtain B β γ v u t m l where B: (B,β) ∈ P
        and lb: P ⊢ β ⇒(m) map Tm u and lc: P ⊢ γ ⇒(l) map Tm t
        and [simp]: α = map Tm v @ Nt B # γ w = v @ u @ t n = m+l
        by blast
      from less.IH[OF _ lc] have c: P' ⊢ γ ⇒* map Tm t by simp
      from assms[THEN le_funD, of B] B
      have β ∈ Ders P' B by (auto simp: Rhss_def)
      then have P' ⊢ [Nt B] ⇒* β by (auto simp: Ders_def)
      from derives_prepend[OF derives_append[OF this]]
      have P' ⊢ α ⇒* map Tm v @ β @ γ by simp
      also from less.IH[OF _ lb] c have P' ⊢ ... ⇒* map Tm w
        by (auto intro!: derives_append_append)
      finally show ?thesis.
    qed
  qed
qed

lemma Lang_Un_redundant: assumes Rhss R ≤ Ders P shows Lang (P ∪ R)
= Lang P
proof (rule antisym)
  show Lang (P ∪ R) ≤ Lang P
    apply (rule Rhss_le_Ders_imp_Lang_le)
    using assms Rhss_le_Ders[of P] by (simp add: le_fun_def Rhss_Un)
```

next
show *Lang P* ≤ *Lang* (*P* ∪ *R*)
apply (*rule le_funI*)
apply (*rule Lang_mono*) **by** *simp*
qed

lemmas *Lang_of_Un_redundant* = *Lang_Un_redundant*[*unfolded Lang_eq_iff_Lang_of_eq*]

Productions whose lhss do not appear in other rules are redundant.

lemma *derive_Un_disj_Lhss*:
assumes α: *Nts_syms* α ∩ *Lhss Q* = {}
shows *P* ∪ *Q* ⊢ α ⇒ β ⟷ *P* ⊢ α ⇒ β
using α **by** (*auto simp*: *Lhss_def derive_iff*)

lemma *deriven_Un_disj_Lhss*:
assumes *PQ*: *Rhs_Nts P* ∩ *Lhss Q* = {} **and** α: *Nts_syms* α ∩ *Lhss Q* = {}
shows *P* ∪ *Q* ⊢ α ⇒(*n*) β ⟷ *P* ⊢ α ⇒(*n*) β (**is** *?l* ⟷ *?r*)
proof
show *?l* ⟹ *?r*
proof (*induction n arbitrary*: β)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
from *Suc.prems* **obtain** β′ **where** *1*: *P* ∪ *Q* ⊢ α ⇒(*n*) β′ **and** *2*: *P* ∪ *Q* ⊢ β′
⇒ β
by (*auto simp*: *relpowp_Suc_right*)
from *Suc.IH*[*OF 1*] **have** *P1*: *P* ⊢ α ⇒(*n*) β′**.**
from *deriven_Nts_syms_subset*[*OF P1*] α *PQ*
have *Nts_syms* β′ ∩ *Lhss Q* = {} **by** *auto*
from *P1 2*[*unfolded derive_Un_disj_Lhss*[*OF this*]]
show *?case* **by** (*auto simp*: *relpowp_Suc_right*)
qed
next
assume *?r*
from *deriven_mono*[*OF _ this*]
show *?l* **by** *auto*
qed

lemma *derives_Un_disj_Lhss*:
assumes *Rhs_Nts P* ∩ *Lhss Q* = {} **and** *Nts_syms* α ∩ *Lhss Q* = {}
shows *P* ∪ *Q* ⊢ α ⇒* β ⟷ *P* ⊢ α ⇒* β
using *deriven_Un_disj_Lhss*[*OF assms*] **by** (*simp add*: *rtranclp_power*)

lemma *Lang_Un_disj_Lhss*:
assumes *disj*: *Rhs_Nts P* ∩ *Lhss Q* = {} **and** *A*: *A* ∉ *Lhss Q*
shows *Lang* (*P* ∪ *Q*) *A* = *Lang P A*
apply (*rule Lang_eqI_derives*)
apply (*rule derives_Un_disj_Lhss*)

using *assms* **by** *auto*

lemma *Lang_disj_Lhss_Un*:
assumes *disj*: *Lhss P* ∩ *Rhs_Nts Q* = {} **and** *A*: *A* ∉ *Lhss P*
shows *Lang* (*P* ∪ *Q*) *A* = *Lang Q A*
using *Lang_Un_disj_Lhss*[*of Q P A*] *assms* **by** (*simp add*: *ac_simps*)

lemma *Lang_of_Un_disj_Lhss*:
assumes *Rhs_Nts P* ∩ *Lhss Q* = {} **and** *Nts_syms* α ∩ *Lhss Q* = {}
shows *Lang_of* (*P* ∪ *Q*) α = *Lang_of P* α
using *derives_Un_disj_Lhss*[*OF assms*] **by** (*simp add*: *Lang_of_def*)

lemma *Lang_of_disj_Lhss_Un*:
assumes *disj*: *Lhss P* ∩ *Rhs_Nts Q* = {} *Nts_syms* α ∩ *Lhss P* = {}
shows *Lang_of* (*P* ∪ *Q*) α = *Lang_of Q* α
using *Lang_of_Un_disj_Lhss*[*of Q P* α] *assms* **by** (*simp add*: *ac_simps*)

lemma *Lang_of_set_Un_disj_Lhss*:
assumes *PQ*: *Rhs_Nts P* ∩ *Lhss Q* = {} **and** *VQ*: ⋃(*Nts_syms* ‘ *V*) ∩ *Lhss Q* = {}
shows *Lang_of_set* (*P* ∪ *Q*) *V* = *Lang_of_set P V*
proof−
{ fix *v* **assume** *v* ∈ *V*
with *VQ* **have** *Nts_syms v* ∩ *Lhss Q* = {} **by** *auto*
note *Lang_of_Un_disj_Lhss*[*OF PQ this*]
}
then show *?thesis* **by** *auto*
qed

lemma *Lang_of_set_disj_Lhss_Un*:
assumes *disj*: *Lhss P* ∩ *Rhs_Nts Q* = {} ⋃(*Nts_syms* ‘ *V*) ∩ *Lhss P* = {}
shows *Lang_of_set* (*P* ∪ *Q*) *V* = *Lang_of_set Q V*
using *Lang_of_set_Un_disj_Lhss*[*of Q P V*] *assms* **by** (*simp add*: *ac_simps*)

1.4 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

1.5 Epsilon-Freeness

Some facts about ε-derivations:

lemma *deriven_Cons_Nil*: $P \vdash x\ \#\ xs \Rightarrow(n)\ [] \longleftrightarrow$
$(\exists A\ \alpha\ l\ m.\ P \vdash \alpha \Rightarrow(l)\ [] \wedge P \vdash xs \Rightarrow(m)\ [] \wedge x = Nt\ A \wedge (A,\alpha) \in P \wedge n = Suc\ (l+m))$
using *deriven_Nt_Cons_map_Tm*[**where** *w=Nil,simplified*]
by (*cases x*, *auto simp add*: *deriven_Nt_Cons_map_Tm*[**where** *w=Nil,simplified*] *deriven_Tm_Cons*)

lemma *derives_Cons_Nil*: $P \vdash x\ \#\ xs \Rightarrow*\ [] \longleftrightarrow$
$(\exists A\ \alpha.\ P \vdash \alpha \Rightarrow*\ [] \wedge P \vdash xs \Rightarrow*\ [] \wedge x = Nt\ A \wedge (A,\alpha) \in P)$
by (*auto simp*: *derives_Cons_decomp*)

Adding production whose rhs does not derive ε by other rules does not change the ε-derivations.

lemma *insert_derives_Nil*:
assumes *α0*: $\neg\ P \vdash \alpha \Rightarrow*\ []$
shows *insert* $(A,\alpha)\ P \vdash \alpha' \Rightarrow*\ [] \longleftrightarrow P \vdash \alpha' \Rightarrow*\ []$ (**is** *?l* $\longleftrightarrow$ *?r*)
proof
assume *?l*
then obtain *n* **where** *insert* $(A,\alpha)\ P \vdash \alpha' \Rightarrow(n)\ []$ **by** (*auto simp*: *rtranclp_power*)
then show $P \vdash \alpha' \Rightarrow*\ []$
proof (*induction n arbitrary*: α' *rule*: *less_induct*)
case (*less n*)

```
    show ?case
    proof (cases α')
      case Nil
      then show ?thesis by simp
    next
      case α': (Cons x xs)
      from less.prems[unfolded α' deriven_Cons_Nil]
      obtain B β l m where β: insert (A,α) P ⊢ β ⇒(l) []
        and xs: insert (A,α) P ⊢ xs ⇒(m) []
        and x: x = Nt B
        and B: (B,β) ∈ insert (A,α) P
        and n: n = Suc (l + m)
        by auto
      from less.IH[OF _ β] have Pβ: P ⊢ β ⇒* [] by (simp add: n)
      from less.IH[OF _ xs] have Pxs: P ⊢ xs ⇒* [] by (simp add: n)
      show ?thesis
      proof (cases (B,β) ∈ P)
        case True
        with Pβ Pxs show ?thesis by (auto simp: α' x derives_Cons_Nil)
      next
        case False
        with B have B = A β = α by auto
        with Pβ α0 show ?thesis by simp
      qed
    qed
  qed
next
  assume r: ?r show ?l by (rule derives_mono[OF _ r], auto)
qed

definition Eps_free where Eps_free R = (∀(_,r) ∈ R. r ≠ [])

abbreviation eps_free rs == Eps_free(set rs)

lemma Eps_freeI:
  assumes ⋀A r. (A,r) ∈ R ⟹ r ≠ [] shows Eps_free R
  using assms by (auto simp: Eps_free_def)

lemma Eps_free_Nil: Eps_free R ⟹ (A,[]) ∉ R
  by (auto simp: Eps_free_def)

lemma Eps_freeE_Cons: Eps_free R ⟹ (A,w) ∈ R ⟹ ∃ 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 ⊢ l ⇒* [] ⟷ l = [] (is ?l ⟷ ?r)
proof
  show ?l ⟹ ?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 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_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: $(A, w) \in P \cup P'$
 and u: $u = u' @ Nt\ A \# v'$
 and v: $v = u' @ w @ v'$
 unfolding *derive_iff* **by** *fast*
then have $(A,w) \notin P'$ **using** *assms(3)* **unfolding** *Nts_syms_def Lhss_def* **by** *auto*
then have $(A,w) \in P$ **using** A_w **by** *blast*
with u v **have** $(A, w) \in P$ **and** u: $u = u' @ Nt\ A \# v'$ **and** v: $v = u' @ w @ v'$ **by** *auto*
then have $P \vdash u \Rightarrow v$ **using** *derive.intros* **by** *fastforce*
moreover have *Nts_syms* $v \cap$ *Lhss* $P' = \{\}$
 using u v *assms* ‹$(A, w) \in P$› **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* $P \cap$ *Lhss* $P' = \{\}$
shows $[\![P \cup P' \vdash u \Rightarrow l(n)\ v;\ \ Nts\_syms\ u \cap Lhss\ P' = \{\}]\!] \Longrightarrow$
  $P \vdash u \Rightarrow l(n)\ v \wedge Nts\_syms\ v \cap Lhss\ P' = \{\}$
proof (*induction n arbitrary*: v)
  case *0*
  then show *?case* **by** *simp*
next
  case (*Suc* n')
  then obtain v' **where** *split*: $P \cup P' \vdash u \Rightarrow l(n')\ v' \wedge P \cup P' \vdash v' \Rightarrow l\ v$
    by (*meson relpowp_Suc_E*)
  with *Suc* **have** $P \vdash u \Rightarrow l(n')\ v' \wedge Nts\_syms\ v' \cap Lhss\ P' = \{\}$
    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* $P \cap$ *Lhss* $P' = \{\}$
shows $[\![P \cup P' \vdash u \Rightarrow(n)\ v;\ \ Nts\_syms\ u \cap Lhss\ P' = \{\}]\!] \Longrightarrow$
  $P \vdash u \Rightarrow(n)\ v \wedge Nts\_syms\ v \cap Lhss\ P' = \{\}$
proof (*induction n arbitrary*: v)
  case *0*
  then show *?case* **by** *simp*
next
  case (*Suc* n')
  then obtain v' **where** *split*: $P \cup P' \vdash u \Rightarrow(n')\ v' \wedge P \cup P' \vdash v' \Rightarrow v$
    by (*meson relpowp_Suc_E*)
  with *Suc* **have** $P \vdash u \Rightarrow(n')\ v' \wedge Nts\_syms\ v' \cap Lhss\ P' = \{\}$
    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 de-
rives_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 de-
rives_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*:
⟦ $A \notin Lhss\ P$; $\{(A,\alpha 1),(A,\alpha 2)\} \cup P \vdash [Nt\ A] \Rightarrow \beta$ ⟧ ⟹ $\beta = \alpha 1 \vee \beta = \alpha 2$
unfolding *derive_singleton* **by**(*auto simp*: *Lhss_def*)

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

lemma *derives_disj_if_derives_fork*:
assumes $A \notin Nts\ P \cup \{B1,B2\}$
and $\{(A,[Nt\ B1]), (A,[Nt\ B2])\} \cup P \vdash [Nt\ A] \Rightarrow* map\ Tm\ w$ (**is** $?P \vdash \_ \Rightarrow* \_$)
shows $P \vdash [Nt\ B1] \Rightarrow* map\ Tm\ w \vee P \vdash [Nt\ B2] \Rightarrow* map\ Tm\ w$
proof −
obtain β **where** *steps*: $?P \vdash [Nt\ A] \Rightarrow \beta\ ?P \vdash \beta \Rightarrow* map\ Tm\ w$
using *converse_rtranclpE*[*OF assms*(*2*)] **by** *blast*
have $\beta = [Nt\ B1] \vee \beta = [Nt\ B2]$
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 $A \notin Nts\ P \cup \{B1,B2\}$
shows $Lang\ (\{(A,[Nt\ B1]),(A,[Nt\ B2])\} \cup P)\ A = (Lang\ P\ B1 \cup Lang\ P\ B2)$
(**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* ⇒ *'t lang*) ⇒ (*'n*, *'t*) *sym* ⇒ *'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* ⇒ *'t lang*) ⇒ (*'n*, *'t*) *syms* ⇒ *'t lang* **where**
inst_syms L w = *concats* (*map* (*inst_sym L*) *w*)

definition *subst_lang* :: (*'n*,*'t*)*Prods* ⇒ (*'n* ⇒ *'t lang*) ⇒ (*'n* ⇒ *'t lang*) **where**
subst_lang P L = (λ*A*. ⋃ *w* ∈ *Rhss P A*. *inst_syms L w*)

definition *Lang_lfp* :: (*'n*, *'t*) *Prods* ⇒ *'n* ⇒ *'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*. ⋃ *i*. *inst_sym* (*L i*) *s*) *xs*) *Ls* = (⋃ *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** $w \in ?l$
with *Cons* **obtain** $u\ v\ i\ j$
where $w = u\ @\ v\ u \in inst\_sym\ (L\ i)\ a\ v \in foldr\ (@@)\ (map\ (inst\_sym\ (L\ j))\ xs)\ Ls$ **by**(*auto*)
then show $w \in ?r$
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 $?r \subseteq ?l$ **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 $L :: nat \Rightarrow 'a \Rightarrow 'b\ lang$
assume *o*: *omega_chain L*
show $(\lambda A.\ \bigcup\ (inst\_syms\ (Sup\ (range\ L))\ `\ Rhss\ P\ A)) = (SUP\ i.\ (\lambda A.\ \bigcup\ (inst\_syms\ (L\ i)\ `\ Rhss\ P\ A)))$
(**is** $(\lambda A.\ ?l\ A) = (\lambda A.\ ?r\ A)$)
proof
fix $A :: 'a$
have $?l\ A = \bigcup(\bigcup i.\ (inst\_syms\ (L\ i)\ `\ Rhss\ P\ A))$
by(*auto simp*: *inst_syms_def inst_sym_Sup_range concats_def foldr_conc_map_inst_sym*[*OF o*])
also have $\ldots = ?r\ A$
by(*auto*)
finally show $?l\ A = ?r\ A$ **.**
qed
qed

theorem *Lang_lfp_SUP*: *Lang_lfp P* = $(SUP\ n.\ ((subst\_lang\ P)\hat{\ }\hat{\ }n)\ (\lambda A.\ \{\}))$
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*: $(\bigwedge A.\ R\ A \subseteq R'\ A) \Longrightarrow w \in inst\_syms\ R\ \alpha \Longrightarrow w \in inst\_syms\ R'\ \alpha$
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* ∈ *subst_lang P L A* ⟹ ∃ α. (*A*,α)∈*P* ∧ *w* ∈ *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 ∈ *inst_syms* (λ*A*. {*w*. *P* ⊢ [*Nt A*] ⇒∗ *map Tm w*}) α ⟹ *P* ⊢ α ⇒∗ *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* ∈ ((*subst_lang P*)^^*n*) (λ*A*. {}) *A* ⟹ *P* ⊢ [*Nt A*] ⇒∗ *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*) (λ*A*. {})
have ∗: *?L A* ⊆ {*w*. *P* ⊢ [*Nt A*] ⇒∗ *map Tm w*} **for** *A*
using *Suc.IH* **by** *blast*
obtain α **where** α: (*A*,α) ∈ *P w* ∈ *inst_syms ?L* α
using *in_subst_langD_inst_syms*[*OF Suc.prems*[*simplified*]] **by** *blast*
show *?case* **using** α(*1*) *derives_if_inst_syms*[*OF inst_syms_mono*[*OF* ∗, *of* _ λ*A*. *A*, *OF* α(*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* (α ! *i*); *length* α = *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** $*$: $\alpha = \alpha 1$ # αr **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*)$\frown\frown$*n*) (λA. {}) *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* $\vdash$ [α ! *i*] $\Rightarrow$(*ms* ! *i*) *map Tm* (*ws* ! *i*))
using *derive_decomp_Tm* **by** *metis*

have $\forall i <$ *length ws*. *ws* ! *i* $\in$ *inst_sym* (λA. ((*subst_lang P*)$\frown\frown$*m*) (λA. {}) *A*) (α ! *i*)
proof (*rule allI* | *rule impI*)+
fix *i*
show *i* < *length ws* $\Longrightarrow$ *ws* ! *i* $\in$ *inst_sym* ((*subst_lang P* $\frown\frown$ *m*) (λA. {}))

```
(α ! i)
      unfolding inst_sym_def
      proof (induction α ! i)
        case (Nt B)
        with ∗ have ∗∗: ms ! i ≤ 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 Expansion of Grammars

```
lemma conc_eq_empty_iff: X @@ Y = {} ⟷ X = {} ∨ Y = {}
```

by *auto*

lemma *Nts_syms_conc*: $\bigcup(Nts\_syms\ `\ (X\ @@\ Y)) =$
$(if\ X = \{\} \vee Y = \{\}\ then\ \{\}\ else\ \bigcup(Nts\_syms\ `\ X) \cup \bigcup(Nts\_syms\ `\ Y))$
by (*force elim*!: *concE*)

We consider the set of admissible expansions of grammars.

For a symbol, one option is not to expand it, and the other option for (expandable) nonterminals is to expand to the all rhss.

definition *Expand_sym_ops* :: ('*n*,'*t*) *Prods* ⇒ '*n set* ⇒ ('*n*,'*t*) *sym* ⇒ ('*n*,'*t*) *syms set set* **where**
Expand_sym_ops P X x =
insert {[*x*]} (*case x of Nt A* ⇒ *if A* ∈ *X then* {*Rhss P A*} *else* {} | _ ⇒ {})

lemma *Expand_sym_ops_simps*:
Expand_sym_ops P X (*Tm a*) = {{[*Tm a*]}}
Expand_sym_ops P X (*Nt A*) = *insert* {[*Nt A*]} (*if A* ∈ *X then* {*Rhss P A*} *else* {})
by (*auto simp*: *Expand_sym_ops_def*)

lemma *Expand_sym_ops_self*: {[*x*]} ∈ *Expand_sym_ops P X x*
by (*simp add*: *Expand_sym_ops_def*)

Mixed words allow all possible combinations of the options for the symbols.

fun *Expand_syms_ops* :: ('*n*,'*t*) *Prods* ⇒ '*n set* ⇒ ('*n*,'*t*) *syms* ⇒ ('*n*,'*t*) *syms set set* **where**
Expand_syms_ops P X [] = {{[]}}
| *Expand_syms_ops P X* (*x*#*xs*) =
{*αs* @@ *βs* | *αs βs*. *αs* ∈ *Expand_sym_ops P X x* ∧ *βs* ∈ *Expand_syms_ops P X xs*}

lemma *Expand_syms_ops_self*: {*α*} ∈ *Expand_syms_ops P X α*
proof (*induction α*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons x α*)
have {*x* # *α*} = {[*x*]} @@ {*α*} **by** (*auto simp*: *insert_conc*)
with *Cons* **show** *?case* **by** (*auto intro*!: *Expand_sym_ops_self*)
qed

lemma *Expand_sym_ops_Lang_of_Cons*:
assumes *U*: *U* ∈ *Expand_sym_ops P X x* **and** *X*: *Lhss Q* ∩ *X* = {}
shows *Lang_of* (*P* ∪ *Q*) (*x*#*xs*) = *Lang_of_set* (*P* ∪ *Q*) *U* @@ *Lang_of* (*P* ∪ *Q*) *xs*
proof−
{ fix *A* **assume** *A* ∈ *X*
with *X* **have** *A* ∉ *Lhss Q* **by** *auto*

```
  then have Rhss (P ∪ Q) A = Rhss P A by (auto simp: Rhss_Un notin_Lhss_iff_Rhss)
   with Lang_of_set_Rhss[of P ∪ Q A]
   have Lang_of_set (P ∪ Q) (Rhss P A) = Lang (P ∪ Q) A by auto
 } note * = this
 from assms
 show ?thesis
  by (auto simp: Expand_sym_ops_simps Lang_of_Cons * split: if_splits sym.splits)
qed

lemma Expand_syms_ops_Lang_of:
 assumes U: U ∈ Expand_syms_ops P L α and L: Lhss Q ∩ L = {}
 shows Lang_of (P ∪ Q) α = Lang_of_set (P ∪ Q) U
proof (insert U, induction α arbitrary: U)
 case Nil
 then show ?case by simp
next
 case (Cons x α)
 from Cons.prems obtain V W
   where U: U = V @@ W
     and V: V ∈ Expand_sym_ops P L x
     and W: W ∈ Expand_syms_ops P L α
   by auto
 show ?case by (simp add: U Cons.IH[OF W] Expand_sym_ops_Lang_of_Cons[OF
V L] Lang_of_set_conc)
qed

definition Expand :: (('n,'t) syms ⇒ ('n,'t) syms set) ⇒ ('n,'t) Prods ⇒ ('n,'t)
Prods where
Expand f P = {(A,α') |A α'. ∃α. (A,α) ∈ P ∧ α' ∈ f α}

lemma Expand_eq_UN[code]:
 Expand f P = (⋃(A,α) ∈ P. Pair A ` f α)
 by (auto simp: Expand_def)

lemma Expand_via_Rhss:
 Expand f P = {(A,α). α ∈ ⋃(f ` Rhss P A)}
 by (auto simp: Expand_def Rhss_def)

lemma Lhss_Expand: Lhss (Expand f P) ⊆ Lhss P
 by (auto simp: Lhss_def Expand_def split: prod.splits)

lemma Rhss_Expand: Rhss (Expand f P) A = ⋃(f ` Rhss P A)
 by (auto simp: Rhss_def Expand_def)

lemma Rhs_Nts_Expand: Rhs_Nts (Expand f P) = (⋃(A,α) ∈ P. ⋃β ∈ f α.
Nts_syms β)
 by (auto simp: Expand_def Rhs_Nts_def)
```

When each production is expanded in an admissible way, then the language is preserved.

definition *Expand_ops* :: ('n,'t) *Prods* ⇒ 'n *set* ⇒ (('n,'t) *syms* ⇒ ('n,'t) *syms set*) *set* **where**
Expand_ops P X = {*f*. ∀α. *f* α ∈ *Expand_syms_ops P X* α}

theorem *Lang_Un_Expand*:
assumes *f*: *f* ∈ *Expand_ops P X* **and** *X*: *X* ∩ *Lhss Q* = {}
shows *Lang* (*P* ∪ *Expand f Q*) = *Lang* (*P* ∪ *Q*)
unfolding *Lang_eq_iff_Lang_of_eq*
proof (*safe intro*!: *ext elim*!: *Lang_ofE_deriven*)
show *P* ∪ *Expand f Q* ⊢ *xs* ⇒(*n*) *map Tm w* ⟹ *w* ∈ *Lang_of* (*P* ∪ *Q*) *xs* **for** *xs w n*
proof (*induction n arbitrary*: *xs w rule*: *less_induct*)
case (*less n*)
show *?case*
proof (*cases* ∃ *A*. *Nt A* ∈ *set xs*)
case *False*
with *less.prems* **have** *xs* = *map Tm w*
by (*metis* (*no_types*, *lifting*) *deriven_from_TmsD destTm.cases ex_map_conv*)
then show *?thesis* **by** (*auto simp*: *Lang_of_map_Tm*)
next
case *True*
then obtain *ys zs A* **where** *xs*: *xs* = *ys* @ *Nt A* # *zs* **by** (*metis split_list*)
from *less.prems*[*unfolded this deriven_Nt_map_Tm*]
obtain δ *m l k v u t* **where** *Aδ*: (*A*,δ) ∈ *P* ∪ *Expand f Q*
and *Lys*: *P* ∪ *Expand f Q* ⊢ *ys* ⇒(*m*) *map Tm v*
and *Lδ*: *P* ∪ *Expand f Q* ⊢ δ ⇒(*l*) *map Tm u*
and *Lzs*: *P* ∪ *Expand f Q* ⊢ *zs* ⇒(*k*) *map Tm t*
and *n*: *n* = *Suc* (*m* + *l* + *k*)
and *w*: *w* = *v* @ *u* @ *t* **by** *force*
from *n* **have** *mn*: *m* < *n* **and** *ln*: *l* < *n* **and** *kn*: *k* < *n* **by** *auto*
with *less.IH Lδ Lys Lzs*
have *u*: *u* ∈ *Lang_of* (*P* ∪ *Q*) δ
and *v*: *v* ∈ *Lang_of* (*P* ∪ *Q*) *ys*
and *t*: *t* ∈ *Lang_of* (*P* ∪ *Q*) *zs* **by** *auto*
show *?thesis*
proof (*cases* (*A*,δ) ∈ *P*)
case *True*
have *Lang_of* (*P* ∪ *Q*) δ ⊆ *Lang* (*P* ∪ *Q*) *A*
apply (*rule Lang_of_prod_subset*)
using *True* **by** *simp*
with *u v t*
show *?thesis* **by** (*auto simp*: *xs w Lang_of_append Lang_of_Nt_Cons*)
next
case *False*
with *Aδ* **obtain** α **where** *AQ*: (*A*,α) ∈ *Q* **and** δ: δ ∈ *f* α **by** (*auto simp*: *Expand_def*)
from *Lδ*[*unfolded* δ] *less.IH*[*of l*] *n*
have *u* ∈ *Lang_of* (*P* ∪ *Q*) δ **by** *auto*
with δ **have** *u* ∈ *Lang_of_set* (*P* ∪ *Q*) (*f* α)

```
        by (auto simp: Lang_of_def)
      also have ... = Lang_of (P ∪ Q) α
        apply (subst Expand_syms_ops_Lang_of)
        using f AQ X by (auto simp: Expand_ops_def)
      also have ... ⊆ Lang (P ∪ Q) A
        apply (rule Lang_of_prod_subset)
        using AQ by auto
      finally have u: u ∈ Lang (P ∪ Q) A by auto
      with v t
      show ?thesis by (simp add: xs w Lang_of_append Lang_of_Nt_Cons)
    qed
   qed
  qed
next
  show P ∪ Q ⊢ xs ⇒(n) map Tm w ⟹ w ∈ Lang_of (P ∪ Expand f Q) xs for
xs w n
  proof (induction n arbitrary: xs w rule: less_induct)
   case (less n)
   show ?case
   proof (cases ∃ A. Nt A ∈ set xs)
    case False
    with less.prems have xs = map Tm w
    by (metis (no_types, lifting) deriven_from_TmsD destTm.cases ex_map_conv)
    then show ?thesis by (auto simp: Lang_of_map_Tm)
   next
    case True
    then obtain ys zs A where xs: xs = ys @ Nt A # zs by (metis split_list)
    from less.prems[unfolded this deriven_Nt_map_Tm]
    obtain α m l k v u t where Aα: (A,α) ∈ P ∪ Q
      and Rys: P ∪ Q ⊢ ys ⇒(m) map Tm v
      and Rα: P ∪ Q ⊢ α ⇒(l) map Tm u
      and Rzs: P ∪ Q ⊢ zs ⇒(k) map Tm t
      and n: n = Suc (m + l + k)
      and w: w = v @ u @ t by force
    from n have mn: m < n and ln: l < n and kn: k < n by auto
    with Rα Rys Rzs
    have u: u ∈ Lang_of (P ∪ Expand f Q) α
      and v: v ∈ Lang_of (P ∪ Expand f Q) ys
      and t: t ∈ Lang_of (P ∪ Expand f Q) zs by (auto simp: less.IH)
    show ?thesis
    proof (cases (A,α) ∈ P)
      case True
      then have (A,α) ∈ P ∪ Expand f Q by simp
      from Lang_of_prod_subset[OF this] u
      have u ∈ Lang (P ∪ Expand f Q) A by auto
      with v w t
      show ?thesis by (auto simp: xs w Lang_of_append Lang_of_Nt_Cons)
    next
      case False
```

with $A\alpha$ **have** $A\alpha Q$: $(A,\alpha) \in Q$ **by** *simp*
with f **have** $f\alpha$: $f\ \alpha \in Expand\_syms\_ops\ P\ X\ \alpha$ **by** (*auto simp*: *Expand_ops_def*)
from X *Lhss_Expand*[*of* $f\ Q$]
have *Lhss*: $Lhss\ (Expand\ f\ Q) \cap X = \{\}$ **by** *auto*
from $X\ A\alpha Q$ **have** *Rhss*: $f\ \alpha \subseteq Rhss\ (Expand\ f\ Q)\ A$
by (*auto simp*: *Rhss_def Expand_def*)
from u *Expand_syms_ops_Lang_of*[*OF* $f\alpha$ *Lhss*]
have $u \in Lang\_of\_set\ (P \cup Expand\ f\ Q)\ (f\ \alpha)$ **by** *auto*
also have $\ldots \subseteq Lang\ (P \cup Expand\ f\ Q)\ A$ **using** *Rhss*
by (*auto simp flip*: *Lang_of_set_Rhss simp*: *Rhss_Un*)
finally have $u \in \ldots$.
with $v\ t$ **show** *?thesis* **by** (*simp add*: *xs w Lang_of_append Lang_of_Nt_Cons*)
qed
qed
qed
qed

corollary *Lang_Expand_Un*:
assumes $f \in Expand\_ops\ P\ X$ **and** $X \cap Lhss\ Q = \{\}$
shows $Lang\ (Expand\ f\ Q \cup P) = Lang\ (Q \cup P)$
using *Lang_Un_Expand*[*OF assms*] **by** (*simp add*: *ac_simps*)

6.1 Instances

For symbols, we just provide a function to expand it.

definition $Expand\_sym :: ('n,'t)\ Prods \Rightarrow 'n\ set \Rightarrow ('n,'t)\ sym \Rightarrow ('n,'t)\ syms\ set$
where
$Expand\_sym\ P\ X\ x = (case\ x\ of\ Nt\ A \Rightarrow if\ A \in X\ then\ Rhss\ P\ A\ else\ \{[x]\} \mid \_ \Rightarrow \{[x]\})$

lemma *Expand_sym_ops*: $Expand\_sym\ P\ L\ x \in Expand\_sym\_ops\ P\ L\ x$
by (*auto simp*: *Expand_sym_def Expand_sym_ops_simps split*: *sym.splits*)

6.1.1 Expanding all nonterminals

fun $Expand\_all\_syms :: ('n,'t)\ Prods \Rightarrow 'n\ set \Rightarrow ('n,'t)\ syms \Rightarrow ('n,'t)\ syms\ set$
where
$Expand\_all\_syms\ P\ X\ [] = \{[]\}$
$\mid Expand\_all\_syms\ P\ X\ (x\#xs) = Expand\_sym\ P\ X\ x\ @@\ Expand\_all\_syms\ P\ X\ xs$

lemma *Expand_all_syms_eq_foldr*:
$Expand\_all\_syms\ P\ X\ \alpha = foldr\ (@@)\ (map\ (Expand\_sym\ P\ X)\ \alpha)\ \{[]\}$
by (*induction* α, *simp_all*)

lemma *Expand_all_syms_append*:
$Expand\_all\_syms\ P\ X\ (\alpha@\beta) = Expand\_all\_syms\ P\ X\ \alpha\ @@\ Expand\_all\_syms\ P\ X\ \beta$

by (*induction* α *arbitrary*: β, *simp_all add*: *conc_assoc*)

lemma *Expand_all_syms_ops*: *Expand_all_syms P X* $\alpha \in$ *Expand_syms_ops P X* α
by (*induction* α, *simp*, *force simp*: *Expand_sym_ops*)

lemma *Expand_all_ops*: *Expand_all_syms P X* $\in$ *Expand_ops P X*
by (*auto simp*: *Expand_ops_def Expand_all_syms_ops*)

abbreviation *Expand_all* :: ('*n*,'*t*) *Prods* $\Rightarrow$ '*n set* $\Rightarrow$ ('*n*,'*t*) *Prods* $\Rightarrow$ ('*n*,'*t*) *Prods*
where
Expand_all P X Q $\equiv$ *Expand* (*Expand_all_syms P X*) *Q*

corollary *Lang_Un_Expand_all*:
assumes *X* $\cap$ *Lhss Q* = {}
shows *Lang* (*P* $\cup$ *Expand_all P X Q*) = *Lang* (*P* $\cup$ *Q*)
using *Lang_Un_Expand*[*OF Expand_all_ops assms*]**.**

corollary *Lang_Expand_all_Un*:
assumes *X* $\cap$ *Lhss Q* = {}
shows *Lang* (*Expand_all P X Q* $\cup$ *P*) = *Lang* (*Q* $\cup$ *P*)
using *Lang_Expand_Un*[*OF Expand_all_ops assms*]**.**

Expand_all removes expanded nonterminals and adds those which the expanded nonterminals depends.

lemma *Expand_all_syms_eq_empty*: *Expand_all_syms P X* α = {} $\longleftrightarrow \neg$ *Nts_syms* $\alpha \cap$ *X* $\subseteq$ *Lhss P*
by (*induction* α)
(*auto simp*: *Expand_sym_def conc_eq_empty_iff notin_Lhss_iff_Rhss*[*symmetric*] *split*: *sym.splits*)

lemma *Nts_syms_Expand_all*:
$\bigcup$(*Nts_syms* ' *Expand_all_syms P X* α) =
(*if Nts_syms* $\alpha \cap$ *X* $\subseteq$ *Lhss P*
then Nts_syms α $-$ *X* $\cup \bigcup$(*Nts_syms* ' $\bigcup$(*Rhss P* ' (*Nts_syms* $\alpha \cap$ *X*)))
else {})
proof (*induction* α)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons x* α)
then show *?case*
by (*auto simp*: *Expand_sym_def Nts_syms_conc Cons Expand_all_syms_eq_empty notin_Lhss_iff_Rhss*[*symmetric*] *split*: *sym.splits*)
qed

lemma *Rhs_Nts_Expand_all*:
Rhs_Nts (*Expand_all P X Q*) =
($\bigcup$(*A*,α) $\in$ *Q*. *if Nts_syms* $\alpha \cap$ *X* $\subseteq$ *Lhss P*

then Nts_syms $\alpha - X \cup \bigcup$(*Nts_syms* ‘ $\bigcup$(*Rhss P* ‘ (*Nts_syms* $\alpha \cap X$)))
else {})
apply (*unfold Rhs_Nts_Expand*)
apply (*rule SUP_cong*[*OF refl*])
apply (*rule prod.case_cong*[*OF refl*])
by (*rule Nts_syms_Expand_all*)

lemma *Rhs_Nts_Expand_all_le*:
Rhs_Nts (*Expand_all P X Q*) $\subseteq$ *Rhs_Nts Q* $- X \cup \bigcup$(*Nts_syms* ‘ $\bigcup$(*Rhss P* ‘ (*Rhs_Nts Q* $\cap X$)))
(**is** *?l* $\subseteq$ *?r*)
proof−
have *?l* $\subseteq$ ($\bigcup$(*A*,α) $\in$ *Q*. *Nts_syms* $\alpha - X \cup \bigcup$(*Nts_syms* ‘ $\bigcup$(*Rhss P* ‘ (*Nts_syms* $\alpha \cap X$))))
apply (*unfold Rhs_Nts_Expand_all*)
apply (*rule SUP_mono′*)
by (*auto simp*: *if_splits*)
also have ... = *?r* **by** (*auto simp*: *Rhs_Nts_def*)
finally show *?thesis*.
qed

Approximately, *Expand_all* depends on nonterminals that are not expanded and those that the expanding grammar depends.

lemma *Rhs_Nts_Expand_all_le_Rhs_Nts*:
Rhs_Nts (*Expand_all P X Q*) $\subseteq$ *Rhs_Nts Q* $- X \cup$ *Rhs_Nts P*
(**is** *?l* $\subseteq$ *?r*)
proof−
note *Rhs_Nts_Expand_all_le*
also have *Rhs_Nts Q* $- X \cup \bigcup$(*Nts_syms* ‘ $\bigcup$(*Rhss P* ‘ (*Rhs_Nts Q* $\cap X$))) $\subseteq$ *?r*
by (*auto simp*: *Rhss_def Rhs_Nts_def*)
finally show *?thesis*.
qed

One can remove a non-recursive part of grammar by expanding others with respect to it.

lemma *Lang_Expand_all_idem*:
assumes *PP*: *Rhs_Nts P* $\cap$ *Lhss P* = {}
and *PQ*: *Lhss P* $\cap$ *Lhss Q* = {} **and** *AP*: *A* $\notin$ *Lhss P*
shows *Lang* (*Expand_all P* (*Lhss P*) *Q*) *A* = *Lang* (*P* $\cup$ *Q*) *A*
(**is** *?l* = *?r*)
proof−
have *?l* = *Lang* (*P* $\cup$ *Expand_all P* (*Lhss P*) *Q*) *A*
apply (*rule Lang_disj_Lhss_Un*[*symmetric*])
using *Rhs_Nts_Expand_all_le_Rhs_Nts*[*of P Lhss P Q*] *PP AP* **by** *auto*
also have ... = *Lang* (*P* $\cup$ *Q*) *A* **by** (*simp add*: *Lang_Un_Expand_all*[*OF PQ*])
finally show *?thesis*.
qed

lemma *Lang_of_Expand_all_idem*:
assumes *PP*: *Rhs_Nts P* ∩ *Lhss P* = {} **and** *PQ*: *Lhss P* ∩ *Lhss Q* = {}
and *AP*: *Nts_syms α* ∩ *Lhss P* = {}
shows *Lang_of* (*Expand_all P* (*Lhss P*) *Q*) *α* = *Lang_of* (*P* ∪ *Q*) *α*
apply (*insert AP*, *induction α*)
using *Lang_Expand_all_idem*[*OF PP PQ*]
by (*simp_all split*: *sym.splits add*: *Lang_of_Cons*)

lemma *Lang_Expand_all_idem_new*:
assumes *PP*: *Rhs_Nts P* ∩ *Lhss P* = {} **and** *PQ*: *Lhss P* ∩ *Lhss Q* = {}
and *AP*: *A* ∈ *Lhss P*
shows *Lang_of_set* (*Expand_all P* (*Lhss P*) *Q*) (*Rhss P A*) = *Lang* (*P* ∪ *Q*) *A*
(**is** *?l* = *?r*)
proof−
have *α* ∈ *Rhss P A* ⟹ *Lang_of* (*Expand_all P* (*Lhss P*) *Q*) *α* = *Lang_of* (*P* ∪ *Q*) *α* **for** *α*
apply (*rule Lang_of_Expand_all_idem*[*OF PP PQ*])
using *PP* **by** (*auto simp*: *Rhss_def Rhs_Nts_def*)
then have *?l* = *Lang_of_set* (*P* ∪ *Q*) (*Rhss P A*) **by** *auto*
moreover have *Rhss P A* = *Rhss* (*P* ∪ *Q*) *A* **using** *AP PQ* **by** (*auto simp*: *Rhss_def dest*: *in_LhssI*)
ultimately show *?thesis* **by** (*simp add*: *Lang_of_set_Rhss*)
qed

lemma *Lang_idem_Un_via_Expand_all*:
assumes *PP*: *Rhs_Nts P* ∩ *Lhss P* = {} **and** *PQ*: *Lhss P* ∩ *Lhss Q* = {}
shows *Lang* (*P* ∪ *Q*) *A* =
(*if A* ∈ *Lhss P then Lang_of_set* (*Expand_all P* (*Lhss P*) *Q*) (*Rhss P A*)
else Lang (*Expand_all P* (*Lhss P*) *Q*) *A*)
using *Lang_Expand_all_idem*[*OF assms*] *Lang_Expand_all_idem_new*[*OF assms*]
by *auto*

corollary *Lang_Un_idem_via_Expand_all*:
assumes *PQ*: *Lhss P* ∩ *Lhss Q* = {} **and** *QQ*: *Rhs_Nts Q* ∩ *Lhss Q* = {}
shows *Lang* (*P* ∪ *Q*) *A* =
(*if A* ∈ *Lhss Q then Lang_of_set* (*Expand_all Q* (*Lhss Q*) *P*) (*Rhss Q A*)
else Lang (*Expand_all Q* (*Lhss Q*) *P*) *A*)
using *Lang_idem_Un_via_Expand_all*[*of Q P A*] *assms* **by** (*auto simp*: *ac_simps*)

6.1.2 Expanding head nonterminals

definition *Expand_hd_syms* :: ('*n*,'*t*) *Prods* ⇒ '*n set* ⇒ ('*n*,'*t*) *syms* ⇒ ('*n*,'*t*) *syms set* **where**
Expand_hd_syms P X α = (*case α of* [] ⇒ {[]} | *x*#*xs* ⇒ *Expand_sym P X x* @@ {*xs*})

lemma *Expand_hd_ops*: *Expand_hd_syms P X* ∈ *Expand_ops P X*
by (*auto simp*: *Expand_hd_syms_def Expand_ops_def intro*!: *Expand_sym_ops Expand_syms_ops_self split*: *list.split*)

abbreviation *Expand_hd* :: ('n,'t) *Prods* ⇒ 'n *set* ⇒ ('n,'t) *Prods* ⇒ ('n,'t) *Prods*
where
Expand_hd P X Q ≡ *Expand* (*Expand_hd_syms P X*) *Q*

theorem *Lang_Expand_hd*:
assumes *X* ∩ *Lhss Q* = {}
shows *Lang* (*Expand_hd P X Q* ∪ *P*) = *Lang* (*Q* ∪ *P*)
using *Lang_Expand_Un*[*OF Expand_hd_ops assms*]**.**

end

7 Replacing Terminals by (Fresh) Nonterminals

Some code setup for partial maps.

declare *dom_empty*[*code_unfold*]

lemma *dom_upd*[*code_unfold*]: *dom* (*f*(*x*↦*y*)) = *insert x* (*dom f*)
by *simp*

value *dom* [*0*::*int* ↦ *10*::*nat*, *1* ↦ *11*, *2* ↦ *12*]

lemma *ranE*: *y* ∈ *ran f* ⟹ (⋀*x*. *f x* = *Some y* ⟹ *thesis*) ⟹ *thesis*
by (*auto simp*: *ran_def*)

lemma *Rhss_image_Pair_inj_on*:
assumes *f*: *inj_on f X* **and** *x*: *x* ∈ *X*
shows *Rhss* ((λ*x*. (*f x*, *g x*)) ‘ *X*) (*f x*) = {*g x*}
using *inj_onD*[*OF f*] *x* **by** (*auto simp*: *Rhss_def*)

First, we define the grammar where fresh nonterminals produce the corresponding terminals. We abstract the partial map from terminals to the fresh nonterminals by *f*.

definition *Replace_Tm_new* :: ('t ⇀ 'n) ⇒ ('n,'t) *Prods* **where**
Replace_Tm_new f = {(*A*,[*Tm a*]) | *A a*. *f a* = *Some A*}

lemma *Replace_Tm_new_code*[*code_unfold*]: *Replace_Tm_new f* = (λ*a*. (*the* (*f a*), [*Tm a*])) ‘ *dom f*
by (*force simp*: *Replace_Tm_new_def*)

value *Replace_Tm_new* [*0*::*int* ↦ *10*::*nat*, *1* ↦ *11*, *2* ↦ *12*]

definition *replace_Tm_new* :: ('t × 'n) *list* ⇒ ('n,'t) *prods* **where**
replace_Tm_new f = [(*A*,[*Tm a*]). (*a*,*A*) ← *f*]

lemma *set_replace_Tm_new*:
distinct (*map fst f*) ⟹ *set* (*replace_Tm_new f*) = *Replace_Tm_new* (*map_of f*)
by (*auto simp*: *replace_Tm_new_def Replace_Tm_new_def*)

Admissible replacements can choose to replace or preserve each terminal.

definition *Replace_Tm_sym_ops* :: $('t \rightharpoonup 'n) \Rightarrow ('n,'t)$ *sym* $\Rightarrow ('n,'t)$ *syms set* **where**
Replace_Tm_sym_ops f x =
insert [*x*] (*case x of Tm a* ⇒ (*case f a of Some A* ⇒ {[*Nt A*]} | _ ⇒ {}) | _ ⇒ {})

fun *Replace_Tm_syms_ops* :: $('t \rightharpoonup 'n) \Rightarrow ('n,'t)$ *syms* $\Rightarrow ('n,'t)$ *syms set* **where**
Replace_Tm_syms_ops f [] = {[]}
| *Replace_Tm_syms_ops f* (*x*#*xs*) =
Replace_Tm_sym_ops f x @@ *Replace_Tm_syms_ops f xs*

definition *Replace_Tm_ops* :: $('t \rightharpoonup 'n) \Rightarrow (('n,'t)$ *syms* $\Rightarrow ('n,'t)$ *syms*$)$ *set* **where**
Replace_Tm_ops f = {*g*. ∀ α. *g* α ∈ *Replace_Tm_syms_ops f* α}

definition *Replace_Tm* :: $('t \rightharpoonup 'n) \Rightarrow (('n,'t)$ *syms* $\Rightarrow ('n,'t)$ *syms*$) \Rightarrow ('n,'t)$ *Prods* $\Rightarrow ('n,'t)$ *Prods* **where**
Replace_Tm f g P = {(*A*, *g* α) | *A* α. (*A*,α) ∈ *P*} ∪ *Replace_Tm_new f*

definition *replace_Tm* :: $('t \times 'n)$ *list* $\Rightarrow (('n,'t)$ *syms* $\Rightarrow ('n,'t)$ *syms*$) \Rightarrow ('n,'t)$ *prods* $\Rightarrow ('n,'t)$ *prods* **where**
replace_Tm f g P = [(*A*, *g* α). (*A*,α) ← *P*] @ *replace_Tm_new f*

lemma *set_replace_Tm*:
distinct (*map fst f*) ⟹ *set* (*replace_Tm f g P*) = *Replace_Tm* (*map_of f*) *g* (*set P*)
by (*auto simp*: *replace_Tm_def Replace_Tm_def set_tms set_replace_Tm_new*)

lemma *Replace_Tm_code*[*code_unfold*]:
Replace_Tm f = (*let Q* = *Replace_Tm_new f in* (λ*g P*. *map_prod id g* ‘ *P* ∪ *Q*))
by (*force simp*: *Replace_Tm_def*)

value *Replace_Tm* [*0*::*int* ↦ *10*::*nat*, *1* ↦ *11*, *2* ↦ *12*] *g P*

Expansion with respect to the grammar *Replace_Tm_new* should revert the fresh nonterminals to the original terminals, while preserving terminals and locked nonterminals.

lemma *Rhss_Replace_Tm_new*:
assumes *inj*: *inj_on f* (*dom f*) **and** *fa*: *f a* = *Some A*
shows *Rhss* (*Replace_Tm_new f*) *A* = {[*Tm a*]}
using *inj_onD*[*OF inj*] *fa*
by (*auto simp add*: *domIff notin_Lhss_iff_Rhss Replace_Tm_new_def Rhss_def*)

lemma *Expand_sym_Replace_Tm_Tm*:
Expand_sym (*Replace_Tm_new f*) *X* (*Tm a*) = {[*Tm a*]}
by (*auto simp*: *Expand_sym_def*)

lemma *Expand_sym_Replace_Tm_Nt*:
assumes *A*: $A \notin X$
shows *Expand_sym* (*Replace_Tm_new f*) *X* (*Nt A*) = {[*Nt A*]}
using *A* **by** (*auto simp*: *Expand_sym_def*)

lemma *Expand_sym_Replace_Tm_new*:
assumes *inj*: *inj_on f* (*dom f*) **and** *a*: *f a* = *Some A* **and** *A*: $A \in X$
shows *Expand_sym* (*Replace_Tm_new f*) *X* (*Nt A*) = {[*Tm a*]}
using *A* **by** (*auto simp*: *Expand_sym_def Rhss_Replace_Tm_new*[*OF inj a*])

lemma *Expand_all_syms_Replace_Tm_ops*:
assumes *inj*: *inj_on f* (*dom f*)
and *X*: *ran f* $\subseteq$ *X* **and** α: $X \cap$ *Nts_syms* $\alpha = \{\}$ $\beta \in$ *Replace_Tm_syms_ops f* α
shows *Expand_all_syms* (*Replace_Tm_new f*) *X* $\beta = \{\alpha\}$
proof (*insert* α, *induction* α *arbitrary*: β)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x* α)
then have $X \cap$ *Nts_syms* $\alpha = \{\}$ **by** *auto*
note *IH* = *Cons.IH*[*OF this*]
show *?case*
proof (*cases x*)
case [*simp*]: (*Nt A*)
from *Cons.prems X* **have** $A \notin X$ **by** *auto*
with *Cons.prems*(*2*)
show *?thesis*
by (*auto simp*: *Replace_Tm_sym_ops_def insert_conc Expand_sym_Replace_Tm_Nt IH*)
next
case [*simp*]: (*Tm a*)
from *X* **have** *AX*: *f a* = *Some A* $\Longrightarrow$ $A \in X$ **for** *A* **by** (*auto simp*: *ranI*)
with *Cons.prems*(*2*)
show *?thesis*
by (*auto simp*: *Replace_Tm_sym_ops_def insert_conc IH*
Expand_sym_Replace_Tm_Tm Expand_sym_Replace_Tm_new[*OF inj*]
split: *option.splits*)
qed
qed

lemma *Expand_all_Replace_Tm_ops*:
assumes *g*: *g* $\in$ *Replace_Tm_ops f* **and** *inj*: *inj_on f* (*dom f*)
and *f*: *ran f* $\subseteq$ *X* **and** *X*: $X \cap$ *Rhs_Nts P* = {}
shows *Expand_all* (*Replace_Tm_new f*) *X* {(*A*, *g* α) | *A* α. (*A*,α) $\in$ *P*} = *P*
proof−
have $*$: (*A*,α) $\in$ *P* $\Longrightarrow$ *Expand_all_syms* (*Replace_Tm_new f*) *X* (*g* α) = {α}
for *A* α
apply (*rule Expand_all_syms_Replace_Tm_ops*[*OF inj f*])

using *X g* **by** (*auto simp*: *Tms_def Rhs_Nts_def Replace_Tm_ops_def*)
then show *?thesis* **by** (*force simp*: *Expand_def*)
qed

Admissible replacements preserves the language, because expanding the replaced grammar results in the original grammar, and expansion preserves the language.

theorem *Lang_Replace_Tm*:
assumes *g*: *g* ∈ *Replace_Tm_ops f*
and *inj*: *inj_on f* (*dom f*)
and *disj*: *ran f* ∩ *Nts P* = {}
and *A*: *A* ∉ *ran f*
shows *Lang* (*Replace_Tm f g P*) *A* = *Lang P A*
(**is** *?l* = *?r*)
proof−
have *?l* = *Lang* ({(*A*, *g* α) | *A* α. (*A*,α) ∈ *P*} ∪ *Replace_Tm_new f*) *A*
by (*simp add*: *Replace_Tm_def*)
also have ... = *Lang* (*Expand_all* (*Replace_Tm_new f*) (*ran f*) {(*A*, *g* α) | *A* α. (*A*,α) ∈ *P*} ∪ *Replace_Tm_new f*) *A*
apply (*subst Lang_Expand_all_Un*)
using *disj* **by** (*auto simp*: *Nts_def Lhss_def*)
also have ... = *Lang* (*P* ∪ *Replace_Tm_new f*) *A*
apply (*subst Expand_all_Replace_Tm_ops*[*OF g inj*])
using *disj*
by (*auto simp*: *Nts_Lhss_Rhs_Nts*)
also have ... = *?r*
apply (*rule Lang_Un_disj_Lhss*) **using** *disj A*
by (*auto simp*: *Replace_Tm_new_def Lhss_Collect Nts_Lhss_Rhs_Nts ran_def*)
finally show *?thesis*.
qed

7.1 Mapping to Fresh Nonterminals

We provide an implementation of a function that maps terminals to corresponding fresh nonterminals.

fun *fresh_map* :: *'a* :: *fresh0 set* ⇒ *'b list* ⇒ *'b* ⇀ *'a* **where**
fresh_map A [] = *Map.empty*
| *fresh_map A* (*x*#*xs*) = (*let a* = *fresh0 A in* (*fresh_map* (*insert a A*) *xs*)(*x* ↦ *a*))

lemma *dom_fresh_map*[*code_unfold*]: *dom* (*fresh_map A xs*) = *set xs*
by (*induction xs arbitrary*: *A*, *simp_all add*: *Let_def*)

fun *fresh_assoc* :: *'a* :: *fresh0 set* ⇒ *'b list* ⇒ (*'b* × *'a*) *list* **where**
fresh_assoc A [] = []
| *fresh_assoc A* (*x*#*xs*) = (*let a* = *fresh0 A in* (*x*,*a*) # *fresh_assoc* (*insert a A*) *xs*)

lemma *map_of_fresh_assoc*: *distinct xs* ⟹ *map_of* (*fresh_assoc A xs*) = *fresh_map A xs*

by (*induction xs arbitrary*: *A*, *auto simp*: *Let_def*)

lemma *map_fst_fresh_assoc*: *map fst* (*fresh_assoc A xs*) = *xs*
by (*induction xs arbitrary*: *A*, *auto simp*: *Let_def*)

lemma *fst_set_fresh_assoc*: *fst ' set* (*fresh_assoc A xs*) = *set xs*
by (*simp flip*: *set_map add*: *map_fst_fresh_assoc*)

lemma *fresh_map_notIn*: *finite A* ⟹ *fresh_map A xs x* = *Some a* ⟹ *a* ∉ *A*
by (*induction xs arbitrary*: *A*; *force simp*: *Let_def fresh0_notIn split*: *if_splits*)

lemma *fresh_map_imp_in*: *fresh_map A xs x* = *Some a* ⟹ *x* ∈ *set xs*
by (*induction xs arbitrary*: *A*; *simp add*: *Let_def split*: *if_splits*)

lemma *fresh_map_disj*: **assumes** *fin*: *finite A* **shows** *ran* (*fresh_map A xs*) ∩ *A* = {}
by (*auto simp*: *fresh_map_notIn*[*OF fin*, *of xs*] *elim*!: *ranE*)

lemma *fresh_map_inj_on*: *finite A* ⟹ *inj_on* (*fresh_map A xs*) (*set xs*)
proof (*induction xs arbitrary*: *A*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons x xs*)
define *a* **where** *a* = *fresh0 A*
from *Cons*
have *IH*: *inj_on* (*fresh_map* (*insert a A*) *xs*) (*set xs*)
and *fin*: *finite* (*insert a A*) **by** *auto*
{ fix *y b* **assume** *y* ∈ *set xs* **and** *fy*: *fresh_map* (*insert a A*) *xs y* = *Some b*
from *fresh_map_notIn*[*OF fin fy*]
have *b* ∉ *A a* ≠ *b* **by** *auto*
} note ∗ = *this this*(*2*)[*symmetric*]
show *?case*
apply (*auto simp flip*: *a_def intro*!: *inj_onI split*: *if_splits simp*: *inj_onD*[*OF IH*])
using ∗(*2*) **apply** *auto*[]
using ∗(*2*) **apply** *metis*
using ∗(*3*) **by** *metis*
qed

lemma *fresh_map_inj_onI*: *finite A* ⟹ *X* ⊆ *set xs* ⟹ *inj_on* (*fresh_map A xs*) *X*
using *inj_on_subset*[*OF fresh_map_inj_on*].

lemma *fresh_map_distinct*:
assumes *fin*: *finite A*
shows *distinct* (*map* (*fresh_map A xs*) *xs*) ⟷ *distinct xs* (**is** *?l* ⟷ *?r*)
using *fresh_map_inj_on*[*OF fin*] **by** (*auto simp*: *distinct_map*)

7.2 Instances

The function replacing a terminal by the corresponding fresh nonterminal is formalized as follows.

definition *replace_Tm_sym* :: $('t \rightharpoonup 'n) \Rightarrow ('n,'t)$ *sym* $\Rightarrow ('n,'t)$ *sym* **where**
replace_Tm_sym f x = (*case x of Tm a* $\Rightarrow$ (*case f a of Some A* $\Rightarrow$ *Nt A* | _ $\Rightarrow$ *x*)
| _ $\Rightarrow$ *x*)

lemma *replace_Tm_sym_simps*:
replace_Tm_sym f (*Nt A*) = *Nt A*
replace_Tm_sym f (*Tm a*) = (*case f a of Some A* $\Rightarrow$ *Nt A* | _ $\Rightarrow$ *Tm a*)
by (*auto simp*: *replace_Tm_sym_def*)

7.2.1 Replacing all terminals

definition *Replace_all_Tm* :: $('t \rightharpoonup 'n) \Rightarrow ('n,'t)$ *Prods* $\Rightarrow ('n,'t)$ *Prods* **where**
[*code_unfold*]: *Replace_all_Tm f* = *Replace_Tm f* (*map* (*replace_Tm_sym f*))

value *Replace_all_Tm* (*fresh_map* {*0*::*nat*,*1*,*2*,*3*} [*10*::*int*,*11*,*12*])
{(*0*,[*Tm 10*, *Tm 12*, *Tm 10*]),(*2*,[*Tm 11*, *Tm 11*, *Tm 12*])}

definition *replace_all_Tm* :: $('t \times 'n)$ *list* $\Rightarrow ('n,'t)$ *prods* $\Rightarrow ('n,'t)$ *prods* **where**
replace_all_Tm f = *replace_Tm f* (*map* (*replace_Tm_sym* (*map_of f*)))

lemma *set_replace_all_Tm*:
distinct (*map fst f*) $\Longrightarrow$ *set* (*replace_all_Tm f P*) = *Replace_all_Tm* (*map_of f*) (*set P*)
by (*simp add*: *replace_all_Tm_def Replace_all_Tm_def set_replace_Tm*)

lemma *map_replace_Tm_sym_ops*:
map (*replace_Tm_sym f*) *xs* $\in$ *Replace_Tm_syms_ops f xs*
by (*induction xs*, *auto split*: *sym.splits option.splits simp*: *Replace_Tm_sym_ops_def insert_conc replace_Tm_sym_simps*)

lemma *map_replace_Tm_ops*:
map (*replace_Tm_sym f*) $\in$ *Replace_Tm_ops f*
by (*simp add*: *Replace_Tm_ops_def map_replace_Tm_sym_ops*)

corollary *Lang_Replace_all_Tm*:
assumes *inj_on f* (*dom f*) *ran f* $\cap$ *Nts P* = {} *A* $\notin$ *ran f*
shows *Lang* (*Replace_all_Tm f P*) *A* = *Lang P A*
using *Lang_Replace_Tm*[*OF map_replace_Tm_ops assms*] **by** (*simp add*: *Replace_all_Tm_def*)

7.2.2 Replacing non-head terminals

definition *replace_Tm_tl_syms* :: $('t \rightharpoonup 'n) \Rightarrow ('n,'t)$ *syms* $\Rightarrow ('n,'t)$ *syms* **where**
replace_Tm_tl_syms f xs = (*case xs of x*#*xs'* $\Rightarrow$ *x* # *map* (*replace_Tm_sym f*) *xs'* | _ $\Rightarrow$ *xs*)

definition *Replace_Tm_tl* :: ('t ⇀ 'n) ⇒ ('n,'t) *Prods* ⇒ ('n,'t) *Prods* **where**
[*code_unfold*]: *Replace_Tm_tl f* = *Replace_Tm f* (*replace_Tm_tl_syms f*)

value *Replace_Tm_tl* (*fresh_map* {*0*::*nat*,*1*,*2*,*3*} [*10*::*int*,*11*,*12*])
{(*0*,[*Tm 10*, *Tm 12*, *Tm 10*]),(*2*,[*Tm 11*, *Tm 11*, *Tm 12*])}

definition *replace_Tm_tl* :: ('t × 'n) *list* ⇒ ('n,'t) *prods* ⇒ ('n,'t) *prods* **where**
replace_Tm_tl f = *replace_Tm f* (*replace_Tm_tl_syms* (*map_of f*))

lemma *set_replace_Tm_tl*:
distinct (*map fst f*) ⟹ *set* (*replace_Tm_tl f P*) = *Replace_Tm_tl* (*map_of f*) (*set P*)
by (*simp add*: *replace_Tm_tl_def Replace_Tm_tl_def set_replace_Tm*)

lemma *replace_Tm_tl_syms_ops*:
replace_Tm_tl_syms f xs ∈ *Replace_Tm_syms_ops f xs*
by (*auto simp*: *Replace_Tm_sym_ops_def replace_Tm_tl_syms_def insert_conc map_replace_Tm_sym_ops split*: *list.splits*)

lemma *replace_Tm_tl_ops*:
replace_Tm_tl_syms f ∈ *Replace_Tm_ops f*
by (*simp add*: *Replace_Tm_ops_def replace_Tm_tl_syms_ops*)

corollary *Lang_Replace_Tm_tl*:
assumes *inj_on f* (*dom f*) *ran f* ∩ *Nts P* = {} *A* ∉ *ran f*
shows *Lang* (*Replace_Tm_tl f P*) *A* = *Lang P A*
using *Lang_Replace_Tm*[*OF replace_Tm_tl_ops assms*] **by** (*simp add*: *Replace_Tm_tl_def*)

end

8 Elimination of Unit Productions

theory *Unit_Elimination*
imports *Context_Free_Grammar*
begin

definition *Unit_prods* :: ('n,'t) *Prods* ⇒ ('n,'t) *Prods* **where**
Unit_prods P = {(*l*,*r*) ∈ *P*. ∃ *A*. *r* = [*Nt A*]}

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

definition *Unit_rm* :: ('n, 't) *Prods* ⇒ ('n, 't) *Prods* **where**
Unit_rm P = *P* − *Unit_prods P*

definition *New_prods* :: ('n, 't) *Prods* ⇒ ('n, 't) *Prods* **where**
New_prods P = {(*A*,*r*). ∃ *B*. (*B*,*r*) ∈ *Unit_rm P* ∧ (*A*, *B*) ∈ *Unit_rtc* (*Unit_prods*

P)}

definition *Unit_elim* :: ('*n*, '*t*) *Prods* ⇒ ('*n*, '*t*) *Prods* **where**
Unit_elim P = *Unit_rm P* ∪ *New_prods P*

definition *Unit_elim_rel* :: ('*n*, '*t*) *Prods* ⇒ ('*n*, '*t*) *Prods* ⇒ *bool* **where**
Unit_elim_rel ps ps' ≡ *ps'* = (*Unit_rm ps* ∪ *New_prods ps*)

corollary *Unit_elim_correct*: *Unit_elim_rel* (*set ps*) (*Unit_elim* (*set ps*))
by (*metis Unit_elim_def Unit_elim_rel_def*)

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 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 ps* **and** *Unit_elim_rel ps ps'*
shows *Eps_free ps'*
using *assms*
unfolding *Unit_elim_rel_def Eps_free_def Unit_rm_def Unit_prods_def New_prods_def*
by *auto*

lemma *Tms_Unit_elim_subset*: *Tms* (*Unit_elim P*) ⊆ *Tms P*
unfolding *Unit_elim_def Unit_rm_def New_prods_def Tms_def* **by**(*auto*)

8.1 Code on lists

definition *unit_prods* :: ('*n*,'*t*) *prods* ⇒ ('*n*,'*t*) *prods* **where**
unit_prods ps = [(*l*,[*Nt A*]). (*l*,[*Nt A*]) ← *ps*]

definition *unit_pairs* :: ('*n*,'*t*) *prods* ⇒ ('*n* × '*n*) *list* **where**
unit_pairs ps = [(*A*,*B*). (*A*,[*Nt B*]) ← *ps*]

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

definition *new_prods* :: ('*n*, '*t*) *prods* ⇒ ('*n*, '*t*) *prods* **where**
new_prods ps = [(*A*,*r*). (*B*,*r*) ← *unit_rm ps*, (*A*,*B'*) ← *trancl_list*(*unit_pairs ps*),
B'=*B*]

definition *unit_elim* :: ('*n*, '*t*) *prods* ⇒ ('*n*, '*t*) *prods* **where**
unit_elim ps = *unit_rm ps* @ *new_prods ps*

lemma *set_unit_prods*: *set* (*unit_prods ps*) = *Unit_prods* (*set ps*)
unfolding *unit_prods_def Unit_prods_def*
by *auto*

lemma *set_unit_rm*: *set* (*unit_rm ps*) = *Unit_rm* (*set ps*)
unfolding *unit_rm_def Unit_rm_def set_minus_list_set set_unit_prods* **..**

lemma *Unit_prods_unit_pairs*[*code*]:
Unit_prods (*set ps*) = *set*(*map* (λ(*A*,*B*). (*A*,[*Nt B*])) (*unit_pairs ps*))
unfolding *Unit_prods_def unit_pairs_def* **by** (*auto*)

lemma *Unit_prods_iff_unit_pairs*:
Unit_prods (*set ps*) ⊢ [*Nt A*] ⇒ [*Nt B*] ⟷ (*A*, *B*) ∈ *set*(*unit_pairs ps*)
unfolding *unit_pairs_def Unit_prods_def* **by**(*auto simp add*: *derive_singleton*)

lemma *Nts_Unit_prods*: (*A*, *B*) ∈ *set*(*unit_pairs ps*)
⟹ *A* ∈ *Lhss* (*Unit_prods* (*set ps*)) ∧ *B* ∈ *Rhs_Nts* (*Unit_prods*(*set ps*))
apply(*auto simp*: *Unit_prods_unit_pairs image_def Nts_Lhss_Rhs_Nts Lhss_def Rhs_Nts_def*
split: *prod.splits*)
apply *blast*
by *force*

lemma *rtc_Unit_prods_if_tc_unit_pairs*:
(*A*,*B*) ∈ *set*(*trancl_list*(*unit_pairs ps*)) ⟹ (*A*,*B*) ∈ *Unit_rtc* (*Unit_prods* (*set ps*))
unfolding *set_trancl_list*
proof(*induction rule*: *converse_trancl_induct*)
case (*base A*)
then show *?case* **unfolding** *Unit_rtc_def*
by (*auto simp add*: *r_into_rtranclp Unit_prods_iff_unit_pairs Nts_Unit_prods Nts_Lhss_Rhs_Nts*)
next
case (*step A A'*)
then show *?case* **unfolding** *Unit_rtc_def*
by (*auto simp add*: *Nts_Lhss_Rhs_Nts Nts_Unit_prods*
intro: *converse_rtranclp_into_rtranclp*[*of derive* (*Unit_prods*(*set ps*))]
Unit_prods_iff_unit_pairs[*THEN iffD2*])
qed

lemma *tc_unit_pairs_if_rtc_Unit_prods*:
fixes *ps* :: ('*n*,'*t*)*prods*
assumes (*A*,*B*) ∈ *Unit_rtc* (*Unit_prods*(*set ps*))
shows *A*=*B* ∨ (*A*,*B*) ∈ *set*(*trancl_list*(*unit_pairs ps*))
proof −
have ∗: *Unit_prods*(*set ps*) ⊢ [*Nt B*] ⇒∗ [*Nt A*] ⟹ *B*=*A* ∨ (*B*,*A*) ∈ (*set*(*unit_pairs ps*))^+ **for** *A B*
proof(*induction* [*Nt B*]::('*n*,'*t*)*syms arbitrary*: *B rule*: *converse_rtranclp_induct*)
case *base* **thus** *?case* **by** *simp*
next
case (*step* α *C*)
from *step.hyps*(*1*) **obtain** *C'* **where** (*C*,*C'*) ∈ *set*(*unit_pairs ps*) α = [*Nt C'*]

by (*auto simp*: *derive_singleton Unit_prods_def unit_pairs_def*)
with *step.hyps*(*2*−)
show *?case*
by (*metis trancl.r_into_trancl trancl_into_trancl2*)
qed
with *assms* **show** *?thesis*
by (*simp add*: *set_trancl_list Unit_rtc_def*)
qed

lemma *Unit_rm_Un_New_prods_eq*: *Unit_rm* (*set ps*) ∪ *New_prods* (*set ps*) = *Unit_rm* (*set ps*) ∪
{(*A*,*r*). ∃ *B*. (*B*,*r*) ∈ *Unit_rm* (*set ps*) ∧ (*A*, *B*) ∈ *set*(*trancl_list*(*unit_pairs ps*))}
unfolding *New_prods_def Unit_rm_def*
by(*auto intro*: *rtc_Unit_prods_if_tc_unit_pairs dest*: *tc_unit_pairs_if_rtc_Unit_prods*)

lemma *Unit_elim_set_code*[*code*]: *Unit_elim* (*set ps*) = *set*(*unit_elim ps*)
unfolding *unit_elim_def Unit_elim_def Unit_rm_Un_New_prods_eq*
by(*auto simp add*: *set_unit_rm new_prods_def*)

corollary *unit_elim_correct*: *Unit_elim_rel* (*set ps*) (*set*(*unit_elim ps*))
by (*metis Unit_elim_set_code Unit_elim_correct*)

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

8.2 Finiteness and Existence

lemma *finiteUnit_prods*: *finite ps* ⟹ *finite* (*Unit_prods ps*)
unfolding *Unit_prods_def*
by (*metis* (*no_types*, *lifting*) *case_prodE finite_subset mem_Collect_eq subsetI*)

definition *NtsCross* :: ('*n*, '*t*) *Prods* ⇒ ('*n* × '*n*) *set* **where**
NtsCross Ps = *Nts Ps* × *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_Unit_rm*: *finite ps* ⟹ *finite* (*Unit_rm ps*)
unfolding *Unit_rm_def* **by** *simp*

lemma *finite_New_prods*: **assumes** *finite ps* **shows** *finite* (*New_prods ps*)
proof −
have *finite* (*Unit_rtc* (*Unit_prods ps*))
using *finiteUnit_prods finite_Unit_rtc assms* **by** *blast*
then show *?thesis*
using *assms finite_Unit_rm npsImage finite_nPSlambda finite_UN* **by** *metis*
qed

lemma *finiteUnit_elim_relRules*: *finite ps* ⟹ *finite* (*Unit_rm ps* ∪ *New_prods ps*)
by (*simp add*: *finite_New_prods finite_Unit_rm*)

lemma *Unit_elim_rel_exists*: *finite ps* ⟹ ∃ *ps'*. *Unit_elim_rel ps ps'* ∧ *finite ps'*
unfolding *Unit_elim_rel_def* **using** *finite_list*[*OF finiteUnit_elim_relRules*] **by** *blast*

lemma *inNonUnitProds*:
p ∈ *Unit_rm ps* ⟹ *p* ∈ *ps*
unfolding *Unit_rm_def* **by** *blast*

lemma *psubDeriv*:
assumes *ps* ⊢ *u* ⇒ *v*
and ∀ *p* ∈ *ps*. *p* ∈ *ps′*
shows *ps′* ⊢ *u* ⇒ *v*
using *assms* **by** (*meson derive_iff*)

lemma *psubRtcDeriv*:
assumes *ps* ⊢ *u* ⇒∗ *v*
and ∀ *p* ∈ *ps*. *p* ∈ *ps′*
shows *ps′* ⊢ *u* ⇒∗ *v*
using *assms* **by** (*induction rule*: *rtranclp.induct*) (*auto simp*: *psubDeriv rtranclp.rtrancl_into_rtrancl*)

lemma *Unit_prods_deriv*:
assumes *Unit_prods ps* ⊢ *u* ⇒∗ *v*
shows *ps* ⊢ *u* ⇒∗ *v*
proof −
have ∀ *p* ∈ *Unit_prods ps*. *p* ∈ *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** *ps′* ⊢ *u* ⇒ *v*
shows *ps* ⊢ *u* ⇒∗ *v*
proof −
obtain *A α r1 r2* **where** *A*: (*A*, *α*) ∈ *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*) ∈ *ps*
using *inNonUnitProds* **by** *blast*
hence *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 *ps* ⊢ [*Nt A*] ⇒∗ [*Nt B*]
    using *Unit_prods_deriv* **by** *blast*
  hence *1*: *ps* ⊢ *r1* @ [*Nt A*] @ *r2* ⇒∗ *r1* @ [*Nt B*] @ *r2*
    using *derives_append derives_prepend* **by** *blast*
  have (*B*, α) ∈ *ps*
    using *B inNonUnitProds* **by** *blast*
  hence *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 *ps'* ⊢ *u* ⇒∗ *v*
   and *Unit_elim_rel ps ps'*
 shows *ps* ⊢ *u* ⇒∗ *v*
 using *assms* **by** (*induction rule*: *rtranclp.induct*) (*auto simp*: *Unit_elim_rel_r3 rtranclp_trans*)

lemma *deriv_Unit_rtc*:
 assumes *ps* ⊢ [*Nt A*] ⇒ [*Nt B*]
 shows (*A*, *B*) ∈ *Unit_rtc* (*Unit_prods ps*)
proof −
 have (*A*, [*Nt B*]) ∈ *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) ∈ *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 *ps* ⊢ [*Nt A*] ⇒ [*Nt B*] *ps'* ⊢ [*Nt B*] ⇒ *v*

shows $ps' \vdash [Nt\ A] \Rightarrow v$
proof −
have *1*: $(A, B) \in Unit\_rtc\ (Unit\_prods\ ps)$
using *deriv_Unit_rtc assms(2)* **by** *fast*
have *2*: $(B, v) \in ps'$
using *assms(3)* **by** (*simp add*: *derive_singleton*)
thus *?thesis*
proof (*cases* $(B, v) \in ps$)
case *True*
hence $(B, v) \in Unit\_rm\ ps$
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 $(B, v) \in New\_prods\ ps$
using *assms(1) 2* **unfolding** *Unit_rm_def Unit_elim_rel_def* **by** *simp*
from *this* **obtain** C **where** C: $(C, v) \in Unit\_rm\ ps \wedge (B, C) \in Unit\_rtc\ (Unit\_prods\ ps)$
unfolding *New_prods_def* **by** *blast*
hence $Unit\_prods\ ps \vdash [Nt\ A] \Rightarrow* [Nt\ C]$
using *1* **unfolding** *Unit_rtc_def* **by** *auto*
hence $(A, C) \in Unit\_rtc\ (Unit\_prods\ ps)$
unfolding *Unit_rtc_def* **using** *1 C Unit_rtc_def* **by** *fastforce*
hence $(A, v) \in New\_prods\ ps$
unfolding *New_prods_def* **using** *C* **by** *blast*
hence $(A, v) \in ps'$
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 $ps \vdash l\ @\ [Nt\ A]\ @\ r \Rightarrow* map\ Tm\ v$
shows $\exists \alpha.\ ps \vdash l\ @\ [Nt\ A]\ @\ r \Rightarrow l\ @\ \alpha\ @\ r \wedge ps \vdash l\ @\ \alpha\ @\ r \Rightarrow* map\ Tm\ v \wedge (A, \alpha) \in ps$
proof −
obtain $l'\ w\ r'$ **where** w: $ps \vdash l \Rightarrow* l' \wedge ps \vdash [Nt\ A] \Rightarrow* w \wedge ps \vdash r \Rightarrow* r' \wedge map\ Tm\ v = l'\ @\ w\ @\ r'$
using *assms(1)* **by** (*metis derives_append_decomp*)
have $Nt\ A \notin set\ (map\ Tm\ v)$
using *assms(1)* **by** *auto*
hence $[Nt\ A] \neq w$
using w **by** *auto*
from *this* **obtain** α **where** α: $ps \vdash [Nt\ A] \Rightarrow \alpha \wedge ps \vdash \alpha \Rightarrow* w$
by (*metis w converse_rtranclpE*)
hence $(A, \alpha) \in ps$

```
    by (simp add: derive_singleton)
  thus ?thesis by (metis α w derive.intros derives_append_decomp)
qed

lemma Unit_elim_rel_r20:
  assumes ps ⊢ u ⇒* map Tm v Unit_elim_rel ps ps'
  shows 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 ps' ⊢ l @ w @ r ⇒* map Tm v ∧ Nt B ∉ set (map Tm v)
      using step.IH assms(2) by auto
    obtain α where α: ps' ⊢ l @ [Nt B] @ r ⇒ l @ α @ r ∧ ps' ⊢ l @ α @ r ⇒*
map Tm v ∧ (B, α) ∈ ps'
      using assms(2) step.IH ‹w=_› Unit_elim_rel_r20_aux[of ps' l B r v] by
blast
    hence (A, α) ∈ ps'
     using assms(2) step.hyps(2) ‹w=_› Unit_elim_rel_r14[of ps ps' A B α] by
(simp add: derive_singleton)
    hence 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) ∈ Unit_rm ps
      unfolding Unit_rm_def using step.hyps(2) by blast
    hence (A, w) ∈ ps'
      using assms(2) unfolding Unit_elim_rel_def  by simp
    hence ps' ⊢ l @ [Nt A] @ r ⇒ 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 P P' ⟹ Lang P' S = Lang P
S
  unfolding Lang_def using Unit_elim_rel_r4 Unit_elim_rel_r20 by blast

corollary Lang_Unit_elim: Lang (Unit_elim (set ps)) A = lang ps A
by (rule Unit_elim_rel_Lang_eq[OF Unit_elim_correct])
```

corollary *lang_unit_elim*: *lang* (*unit_elim ps*) *A* = *lang ps A*
by (*metis unit_elim_correct Unit_elim_rel_Lang_eq*)

end

9 Elimination of Epsilon Productions

theory *Epsilon_Elimination*
imports
Context_Free_Grammar
HOL−Library.While_Combinator
begin

inductive *Nullable* :: ('*n*,'*t*) *Prods* ⇒ ('*n*,'*t*) *sym* ⇒ *bool*
for *P* **where**
NullableSym:
⟦ (*A*, *w*) ∈ *P*; ∀ *s* ∈ *set w*. *Nullable P s*⟧
⟹ *Nullable P* (*Nt A*)

abbreviation *Nullables P w* ≡ (∀ *s* ∈ *set w*. *Nullable P s*)

definition *nullables_wrt* :: '*n set* ⇒ ('*n*, '*t*) *syms* ⇒ *bool* **where**
nullables_wrt N w = (*Tms_syms w* = {} ∧ *Nts_syms w* ⊆ *N*)

definition *nullable_step* :: ('*n*,'*t*)*Prods* ⇒ '*n set* ⇒ '*n set* **where**
nullable_step P N = *fst* ' {(*A*,*w*) ∈ *P*. *nullables_wrt N w*}

definition *nullable_fun* :: ('*n*,'*t*)*Prods* ⇒ '*n set option* **where**
nullable_fun P = *while_option* (λ*N*. *nullable_step P N* ≠ *N*) (*nullable_step P*)
{}

lemma *nullable_fun* {(*0*::*int*,[*Nt 1*::(*int*,*int*)*sym*]), (*1*,[])} = *Some*{*0*,*1*}
by *eval*

lemma *mono_nullable_step*: *mono* (*nullable_step P*)
unfolding *mono_def nullable_step_def nullables_wrt_def* **by** *auto*

lemma *while_option_Some_closed*:
fixes *f* :: '*a*::*complete_lattice* ⇒ '*a*
assumes *while_option* (λ*x*. *f x* ≠ *x*) *f bot* = *Some x* **shows** *f x* = *x*
using *while_option_stop2*[*OF assms*(*1*)] **by** *fastforce*

lemma *nullable_fun_Some_closed*: *nullable_fun P* = *Some N* ⟹ *nullable_step P N* = *N*
unfolding *nullable_fun_def* **using** *while_option_Some_closed*[*of nullable_step P*] **by** *blast*

lemma *Nullable_if_nullable_fun*:
assumes *finite P nullable_fun P = Some N* **shows** $\forall A{\in}N.$ *Nullable P (Nt A)*
proof −
let *?I* = $\lambda N.\ \forall A{\in}N.$ *Nullable P (Nt A)*
have *0*: *?I* {} **by** *simp*
have *?I* (*nullable_step P N*) **if** *?I N* **for** *N*
proof −
have *Nullable P (Nt A)* **if** *asms*: (*A*, *w*) ∈ *P Tms_syms w* = {} *Nts_syms w* ⊆ *N* **for** *A w*
proof −
have *Nullable P s* **if** *s* ∈ *set w* **for** *s*
proof −
have *s* ∈ *Nt* ‘ *N* **using** *asms*(*2*,*3*) ‹*s* ∈ *set w*› **unfolding** *Nts_syms_def Tms_syms_def*
by(*cases s*) *auto*
thus *?thesis* **using** ‹*?I N*› **by** *blast*
qed
then show *?thesis* **by** (*metis asms*(*1*) *NullableSym*)
qed
then show *?thesis* **using** ‹*?I N*› **unfolding** *nullable_step_def nullables_wrt_def* **by** *auto*
qed
from *while_option_rule*[**where** *P* = *?I* **and** *s* = {}, *OF this assms*(*2*)[*unfolded nullable_fun_def*] *0*]
show *?thesis* **by** *blast*
qed

lemma *nullable_fun_if_Nullable*: **assumes** *nullable_fun P = Some N*
shows *Nullable P s* ⟹ (⋀*A*. *s* = *Nt A* ⟹ *A* ∈ *N*)
proof(*induction rule*: *Nullable.induct*)
case (*NullableSym B w*)
then have [*simp*]: *B* = *A* **by** *auto*
from *NullableSym* **have** *A* ∈ *nullable_step P N*
unfolding *nullable_step_def nullables_wrt_def image_def Nts_syms_def Tms_syms_def*
apply *auto*
using *Nullable.cases* **by** *blast*
with *NullableSym* **show** *?case*
using *assms nullable_fun_Some_closed* **by** *blast*
qed

lemma *nullable_fun_Some*: **assumes** *finite P* **shows** ∃ *N*. *nullable_fun P = Some N*
proof −
let *?M* = *Nts P*
have ∗: ⋀*X*. *X* ⊆ *Nts P* ⟹ *nullable_step P X* ⊆ *Nts P*
unfolding *nullable_step_def nullables_wrt_def Nts_def* **by**(*auto*)
from *while_option_finite_subset_Some*[*OF mono_nullable_step* ∗ *finite_Nts*[*OF assms*]]
show *?thesis* **unfolding** *nullable_fun_def* **by** *blast*

qed

lemma *Nullable_iff_nullable_fun*: *finite P* $\Longrightarrow$ *Nullable P* (*Nt A*) = (*A* $\in$ *the*(*nullable_fun P*))
by (*metis Nullable_if_nullable_fun nullable_fun_Some nullable_fun_if_Nullable option.sel*)

lemma *nullable_Tm*[*code*]: *Nullable P* (*Tm a*) = *False*
using *Nullable.cases* **by** *blast*

lemma *nullable_Nt*[*code*]: *Nullable* (*set ps*) (*Nt A*) = (*A* $\in$ *the*(*nullable_fun* (*set ps*)))
by (*simp add*: *Nullable_iff_nullable_fun*)

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

lemma *Nullable* {(*0*::*int*,[*Nt 1*::(*int*,*int*)*sym*]), (*1*,[])} (*Nt 0*)
by *eval*

lemma *nullables_if*:
assumes *P* $\vdash$ *u* $\Rightarrow*$ *v*
and *u*=[*a*] *Nullables P v*
shows *Nullables P 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 ‹*P* $\vdash$ *v* $\Rightarrow$ *w*› **obtain** *A* α *l r* **where** *A*α: *v* = *l* @ [*Nt A*] @ *r* $\wedge$ *w* = *l* @ α @ *r* $\wedge$ (*A*, α) $\in$ *P*
by (*auto simp*: *derive.simps*)
from *this* ‹*Nullables P w*› **have** *Nullables P* α $\wedge$ *Nullables P l* $\wedge$ *Nullables P r*
by *simp*
hence *Nullables P* [*Nt A*]
using *A*α *Nullable.simps* **by** *auto*
from *this* ‹*Nullables P* α $\wedge$ *Nullables P l* $\wedge$ *Nullables P r*› **have** *Nullables P v*
using *A*α **by** *auto*
thus *?case*
using *rtrancl_into_rtrancl.IH rtrancl_into_rtrancl.prems*(*1*) **by** *blast*
qed

lemma *nullable_if*: *P* $\vdash$ [*a*] $\Rightarrow*$ [] $\Longrightarrow$ *Nullable P a*
using *nullables_if*[*of P* [*a*] [] *a*] **by** *simp*

lemma *nullable_aux*: $\forall$ *s*$\in$*set gamma*. *P* $\vdash$ [*s*] $\Rightarrow*$ [] $\Longrightarrow$ *P* $\vdash$ *gamma* $\Rightarrow*$ []

proof (*induction gamma*)
  case (*Cons a list*)
  hence $P \vdash list \Rightarrow* []$
    by *simp*
  moreover have $P \vdash [a] \Rightarrow* []$
    using *Cons* **by** *simp*
  ultimately show *?case*
    using *derives_Cons*[*of* ‹P› *list* ‹$[]$› ‹a›] **by** *simp*
qed *simp*

lemma *if_nullable*: $Nullable\ P\ a \Longrightarrow P \vdash [a] \Rightarrow* []$
proof (*induction rule*: *Nullable.induct*)
  case (*NullableSym x gamma*)
    hence $P \vdash [Nt\ x] \Rightarrow* gamma$
      using *derive_singleton* **by** *blast*
    also have $P \vdash gamma \Rightarrow* []$
      using *NullableSym nullable_aux* **by** *blast*
  finally show *?case* **.**
qed

corollary *nullable_iff*: $Nullable\ P\ a \longleftrightarrow P \vdash [a] \Rightarrow* []$
  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 P = {(*l*,*r*'). ∃ *r*. (*l*,*r*) ∈ *P* ∧ *r*' ∈ *set* (*eps_closure P r*) ∧ (*r*' ≠ [])}

lemma *Eps_elim_code*[*code*]: *Eps_elim P* =
  (⋃(*l*,*r*) ∈ *P*. ⋃*r*' ∈ *set* (*eps_closure P r*). *if r*' = [] *then* {} *else* {(*l*,*r*')})
unfolding *Eps_elim_def* **by** (*auto split*: *prod.split*)

definition *eps_elim* :: ('*n*, '*t*) *prods* ⇒ ('*n*, '*t*) *prods* **where**
eps_elim ps ≡ *concat* (*map* (λ(*l*,*r*). *map* (λ*r*'. (*l*,*r*')) (*filter* (λ*r*'. *r*' ≠ []) (*eps_closure* (*set ps*) *r*))) *ps*)

lemma *set_eps_elim*: *set*(*eps_elim ps*) = *Eps_elim* (*set ps*)
unfolding *eps_elim_def Eps_elim_def* **by** *auto*

lemma *Eps_elim*
  {(*0*::*int*, [*Nt 1*, *Nt 2*, *Nt 1*]), (*1*, [*Tm* (*0*::*int*), *Nt 1*]), (*1*, []), (*2*, [*Tm 1*, *Nt 2*]), (*2*,[])}
 = {(*2*, [*Tm 1*, *Nt 2*]), (*2*, [*Tm 1*]), (*1*, [*Tm 0*, *Nt 1*]), (*1*, [*Tm 0*]), (*0*, [*Nt 1*, *Nt 2*, *Nt 1*]),
    (*0*, [*Nt 1*, *Nt 2*]), (*0*, [*Nt 1*, *Nt 1*]), (*0*, [*Nt 1*]), (*0*, [*Nt 2*, *Nt 1*]), (*0*, [*Nt 2*])}

by *eval*

lemma *Eps_free_Eps_elim*: *Eps_free* (*Eps_elim ps'*)
unfolding *Eps_elim_def Eps_free_def* **by** *blast*

Eps_elim is identity on *Eps_free* input.

lemma *Eps_free_not_Nullable*: *Eps_free P* $\Longrightarrow \neg$ *Nullable P A*
by (*auto simp*: *nullable_iff Eps_free_derives_Nil*)

lemma *Eps_free_eps_closure*: *Eps_free P* $\Longrightarrow$ *eps_closure P w* = [*w*]
by (*induction w*, *auto simp*: *Eps_free_not_Nullable*)

lemma *Eps_elim_id*: *Eps_free P* $\Longrightarrow$ *Eps_elim P* = *P*
by (*auto simp*: *Eps_elim_def Eps_free_eps_closure Eps_free_Nil*)

definition *Eps_elim_fun* :: ('*n*, '*t*) *Prods* $\Rightarrow$ ('*n*, '*t*) *prod* $\Rightarrow$ ('*n*, '*t*) *Prods* **where**

Eps_elim_fun ps p = {(*l'*,*r'*). *l'* = *fst p* $\wedge$ *r'* $\in$ *set* (*eps_closure ps* (*snd p*)) $\wedge$ (*r'* $\neq$ []) }

lemma *Eps_elim_fun_eq*: *Eps_elim ps* = $\bigcup$((*Eps_elim_fun ps*) ' *ps*)
proof
show *Eps_elim ps* $\subseteq$ ($\bigcup$ (*Eps_elim_fun ps* ' *ps*))
unfolding *Eps_elim_def Eps_elim_fun_def* **by** *auto*
next
show $\bigcup$((*Eps_elim_fun ps*) ' *ps*) $\subseteq$ *Eps_elim ps*
proof
fix *x*
assume *x* $\in$ $\bigcup$((*Eps_elim_fun ps*) ' *ps*)
obtain *l r'* **where** *x* = (*l*,*r'*) **by** *fastforce*
hence (*l*,*r'*) $\in$ $\bigcup$((*Eps_elim_fun ps*) ' *ps*)
using ‹*x* $\in$ $\bigcup$((*Eps_elim_fun ps*) ' *ps*)› **by** *simp*
hence *1*: $\exists$ *r*. *r'* $\in$ *set* (*eps_closure ps r*) $\wedge$ (*r'* $\neq$ []) $\wedge$ (*l*,*r*) $\in$ *ps*
using *Eps_elim_fun_def* **by** *fastforce*
from *this* **obtain** *r* **where** *r'* $\in$ *set* (*eps_closure ps r*) $\wedge$ (*l*,*r*) $\in$ *ps*
by *blast*
thus *x* $\in$ *Eps_elim ps* **unfolding** *Eps_elim_fun_def Eps_elim_def*
using *1* ‹*x* = (*l*, *r'*)› **by** *blast*
qed
qed

lemma *finite_Eps_elim*: **assumes** *finite ps* **shows** *finite* (*Eps_elim ps*)
proof −
have $\forall$ *p* $\in$ *ps*. *finite* (*Eps_elim_fun ps p*)
unfolding *Eps_elim_fun_def* **by** *auto*
hence *finite* ($\bigcup$((*Eps_elim_fun ps*) ' *ps*))
using *finite_UN assms* **by** *simp*
thus *?thesis* **using** *Eps_elim_fun_eq* **by** *metis*

qed

lemma *eps_closure_nullable*: $[] \in set\ (eps\_closure\ ps\ w) \Longrightarrow Nullables\ ps\ w$
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*: $r' \in set\ (eps\_closure\ ps\ r) \Longrightarrow ps \vdash r \Rightarrow* r'$
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*: $ps \vdash r \Rightarrow* e$
      using *Cons.IH* **by** *blast*
    hence *2*: $ps \vdash [a]@r \Rightarrow* [a]@e$
      using *e derives_prepend* **by** *blast*
    have $ps \vdash [a] \Rightarrow* []$
      using *True if_nullable* **by** *blast*
    hence $ps \vdash [a]@r \Rightarrow* r$
      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 $ps \vdash r \Rightarrow* e$
      using *Cons.IH* **by** *simp*
    hence $ps \vdash [a]@r \Rightarrow* [a]@e$
      using *derives_prepend* **by** *blast*
    thus *?thesis*
      using *e* **by** *simp*
  qed
qed *simp*

lemma *Eps_elim_rel_r2*:
  assumes $Eps\_elim\ ps \vdash u \Rightarrow v$
  shows $ps \vdash u \Rightarrow* v$
  using *assms*

proof −
obtain $A\ \alpha\ x\ y$ **where** A: $(A,\ \alpha) \in Eps\_elim\ ps \wedge u = x\ @\ [Nt\ A]\ @\ y \wedge v = x\ @\ \alpha\ @\ y$
using *assms derive.cases* **by** *meson*
hence *1*: $(A,\ \alpha) \in \{(l,r').\ \exists\, r.\ (l,r) \in ps \wedge r' \in set\ (eps\_closure\ ps\ r) \wedge (r' \neq [])\}$
unfolding *Eps_elim_def* **by** *simp*
obtain r **where** r: $(A,\ r) \in ps \wedge \alpha \in set\ (eps\_closure\ ps\ r)$
using *1* **by** *blast*
hence $ps \vdash r \Rightarrow* \alpha$
using *Eps_elim_rel_1* **by** *blast*
hence *2*: $ps \vdash x\ @\ r\ @\ y \Rightarrow* x\ @\ \alpha\ @\ y$
using *r derives_prepend derives_append* **by** *blast*
hence $ps \vdash x\ @\ [Nt\ A]\ @\ y \Rightarrow x\ @\ r\ @\ y$
using *r derive.simps* **by** *fast*
thus *?thesis*
using *2* **by** (*simp add*: *A*)
qed

lemma *Eps_elim_rel_r3*:
assumes $Eps\_elim\ ps \vdash u \Rightarrow* v$
shows $ps \vdash u \Rightarrow* v$
using *assms* **by** (*induction v rule*: *rtranclp_induct*) (*auto simp*: *Eps_elim_rel_r2 rtranclp_trans*)

lemma *Eps_elim_rel_r5*: $r \in set\ (eps\_closure\ ps\ r)$
by (*induction r*) *auto*

lemma *Eps_elim_rel_r4*:
assumes $(l,r) \in ps$
and $(r' \neq [])$
and $r' \in set\ (eps\_closure\ ps\ r)$
shows $(l,r') \in Eps\_elim\ ps$
using *assms* **unfolding** *Eps_elim_def* **by** *blast*

lemma *Eps_elim_rel_r7*:
assumes $ps \vdash [Nt\ A] \Rightarrow v$
and $v' \in set\ (eps\_closure\ ps\ v) \wedge (v' \neq [])$
shows $Eps\_elim\ ps \vdash [Nt\ A] \Rightarrow v'$
proof −
have $(A,v) \in ps$
using *assms*(*1*) **by** (*simp add*: *derive_singleton*)
hence $(A,v') \in Eps\_elim\ ps$
using *assms Eps_elim_rel_r4 conjE* **by** *fastforce*
thus *?thesis*
using *derive_singleton* **by** *fast*
qed

lemma *Eps_elim_rel_r12a*:

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

lemma *Eps_elim_rel_r12b*:
assumes $x' \in set\ (eps\_closure\ ps\ x)$
and $y' \in set\ (eps\_closure\ ps\ y)$
and $z' \in set\ (eps\_closure\ ps\ z)$
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 ps ⊢ [Nt S] ⇒* u
    and v ∈ set (eps_closure ps u) ∧ (v ≠ [])
  shows Eps_elim ps ⊢ [Nt S] ⇒* 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: Eps_elim ps ⊢ [Nt S] ⇒* (x'@[Nt A]@y')
        using step by blast
      have ps ⊢ [Nt A] ⇒ w
        using step(2) derive_singleton by blast
      hence Eps_elim ps ⊢ [Nt A] ⇒ w'
        using Eps_elim_rel_r7[of ps A w w'] False step v by blast
      hence Eps_elim 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(1))
```

```
  qed
qed

theorem Eps_elim_rel_eq_if_noe:
  assumes [] ∉ Lang ps S
  shows Lang ps S = Lang (Eps_elim ps) S
proof
  show Lang ps S ⊆ Lang (Eps_elim ps) S
  proof
    fix x
    assume x ∈ Lang ps S
    have ∀ x. 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 Eps_elim 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 (Eps_elim ps) S
      using Lang_def ‹x ∈ Lang ps S› by fast
  qed
next
  show Lang (Eps_elim ps) S ⊆ Lang ps S
  proof
    fix x'
    assume x' ∈ Lang (Eps_elim ps) S
    show x' ∈ Lang ps S
      using assms Lang_def ‹x' ∈ Lang (Eps_elim ps) S› Eps_elim_rel_r3[of ps
‹[Nt S]› ‹map Tm x'›] 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: [] ∉ Lang (Eps_elim ps) S
proof (rule notI)
  assume [] ∈ Lang (Eps_elim ps) S
  hence Eps_elim ps ⊢ [Nt S] ⇒* map Tm []
    using Lang_def by fast
  hence Eps_elim ps ⊢ [Nt S] ⇒* []
    by simp
  hence ∃ A. Eps_elim ps ⊢ [Nt S] ⇒* [Nt A] ∧ (A, []) ∈ Eps_elim ps
    using noe_lang_Eps_elim_rel_aux[of ‹Eps_elim ps›] by blast
  thus False
    unfolding Eps_elim_def by blast
```

qed

theorem *Lang_Eps_elim*: *Lang* (*Eps_elim ps*) *S* = *Lang ps S* − {[]}
proof
  show *Lang* (*Eps_elim ps*) *S* ⊆ *Lang ps S* − {[]}
  proof
    fix *w*
    assume *w* ∈ *Lang* (*Eps_elim ps*) *S*
    hence *w* ∈ *Lang* (*Eps_elim ps*) *S* − {[]}
      by (*simp add*: *noe_lang_Eps_elim_rel*)
    thus *w* ∈ *Lang ps S* − {[]}
      by (*auto simp*: *Lang_def Eps_elim_rel_r3*)
  qed
next
  show *Lang ps S* − {[]} ⊆ *Lang* (*Eps_elim ps*) *S*
  proof
    fix *w*
    assume *w* ∈ *Lang ps S* − {[]}
    hence *1*: (*map Tm w*) ≠ []
      by *simp*
    have *2*: *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 *Eps_elim ps* ⊢ [*Nt S*] ⇒∗ (*map Tm w*)
      using *1 2 Eps_elim_rel_r15*[*of ps*] **by** *simp*
    thus *w* ∈ *Lang* (*Eps_elim ps*) *S*
      by (*simp add*: *Lang_def*)
  qed
qed

lemma *set_eps_closure_subset*: *u* ∈ *set*(*eps_closure P w*) ⟹ *set u* ⊆ *set w*
apply(*induction P w arbitrary*: *u rule*: *eps_closure.induct*)
 apply *simp*
apply (*fastforce split*: *if_splits*)
done

lemma *Lhss_Eps_elim*: *Lhss* (*Eps_elim P*) ⊆ *Lhss P*
by(*auto simp*: *Lhss_def Eps_elim_def dest*: *set_eps_closure_subset*)

lemma *Tms_Eps_elim*: *Tms* (*Eps_elim P*) ⊆ *Tms P*
by(*auto simp*: *Tms_def Tms_syms_def Eps_elim_def dest*: *set_eps_closure_subset*)

lemma *Rhs_Nts_Eps_elim*: *Rhs_Nts* (*Eps_elim P*) ⊆ *Rhs_Nts P*
by(*auto simp*: *Rhs_Nts_def Nts_syms_def Eps_elim_def dest*: *set_eps_closure_subset*)

lemma *Nts_Eps_elim*: *Nts* (*Eps_elim P*) ⊆ *Nts P*
  using *Lhss_Eps_elim*[*of P*] *Rhs_Nts_Eps_elim*[*of P*]
by (*auto simp*: *Nts_Lhss_Rhs_Nts*)

corollary *nts_eps_elim*: *Nts*(*set*(*eps_elim ps*)) ⊆ *Nts*(*set ps*)
by (*metis set_eps_elim Nts_Eps_elim*)

corollary *lang_eps_elim*: *lang* (*eps_elim ps*) *S* = *lang ps S* − {[]}
by (*metis Lang_Eps_elim set_eps_elim*)

corollary *eps_free_eps_elim*: *eps_free* (*eps_elim ps*)
by (*metis set_eps_elim Eps_free_Eps_elim*)

end

10 Conversion to Chomsky Normal Form

theory *Chomsky_Normal_Form*
imports
Unit_Elimination
Epsilon_Elimination
Replace_Terminals
begin

The conversion to Chomsky Normal Form (CNF) is achieved by, in that order, epsilon and unit elimination, uniformization and binarization. A production $A \to \alpha$ is

uniform if α contains no terminal unless *length* α = *1*,

binary if *length* $\alpha \leq$ *2*.

The start symbol S is passed around explicitly to avoid generating S as a fresh name. Of course the nonterminals in the productions *ps* are avoided. However, if $S \notin$ *Nts* (*set ps*) or *lang ps S* = {[]} (in which case epsilon elimination eliminates S), S could accidentally be generated as a fresh name. One could perform the CNF conversion without avoiding S explicitly. As a result one would get a CNF that is independent of S (in contrast to now), but would need to add the preconditions $S \in$ *Nts* (*set ps*) and *lang ps S* ≠ {[]}, which would also be inherited by any application of the CNF conversion.

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

10.1 Uniformization

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

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

Bad_tms P = (⋃(*A*,*α*) ∈ *P*. *if length α* ≥ *2 then Tms_syms α else* {})

definition *bad_tms* :: (*'n*,*'t*) *prods* ⇒ *'t list* **where**
bad_tms ps = *remdups*(*concat* ((*map tms_syms o filter* (λ*u*. *length u* ≥ *2*) *o map snd*) *ps*))

lemma *set_bad_tms*: *set*(*bad_tms ps*) = *Bad_tms* (*set ps*)
unfolding *Bad_tms_def bad_tms_def*
by (*auto simp*: *set_tms_syms split*: *if_splits*)

definition *replace_Tm_2_syms* **where**
replace_Tm_2_syms f xs = (*if length xs* < *2 then xs else map* (*replace_Tm_sym f*) *xs*)

definition *Replace_Tm_2* :: (*'t* ⇀ *'n*) ⇒ (*'n*,*'t*) *Prods* ⇒ (*'n*,*'t*) *Prods* **where**
[*code_unfold*]: *Replace_Tm_2 f* = *Replace_Tm f* (*replace_Tm_2_syms f*)

definition *replace_Tm_2* :: (*'t* × *'n*) *list* ⇒ (*'n*,*'t*) *prods* ⇒ (*'n*,*'t*) *prods* **where**
replace_Tm_2 f = *replace_Tm f* (*replace_Tm_2_syms* (*map_of f*))

lemma *set_replace_Tm_2*:
distinct (*map fst f*) ⟹ *set* (*replace_Tm_2 f ps*) = *Replace_Tm_2* (*map_of f*) (*set ps*)
by (*auto simp add*: *replace_Tm_2_def Replace_Tm_2_def set_replace_Tm*)

lemma *replace_Tm_2_syms_ops*:
replace_Tm_2_syms f α ∈ *Replace_Tm_syms_ops f α*
proof (*cases length α* < *2*)
case *False*
thus *?thesis*
by (*simp add*: *replace_Tm_2_syms_def map_replace_Tm_sym_ops*)
next
case *True*
thus *?thesis*
by(*cases α*)
(*auto simp*: *replace_Tm_2_syms_def Replace_Tm_sym_ops_def*)
qed

lemma *replace_Tm_2_ops*:
replace_Tm_2_syms f ∈ *Replace_Tm_ops f*
by (*simp add*: *Replace_Tm_ops_def replace_Tm_2_syms_ops*)

corollary *Lang_Replace_Tm_2*:
assumes *inj_on f* (*dom f*) *ran f* ∩ *Nts P* = {} *A* ∉ *ran f*
shows *Lang* (*Replace_Tm_2 f P*) *A* = *Lang P A*
using *Lang_Replace_Tm*[*OF replace_Tm_2_ops assms*] **by** (*simp add*: *Replace_Tm_2_def*)

corollary *lang_replace_Tm_2*:

```
  assumes dist: distinct (map fst f)
    and inj: inj_on (map_of f) (fst ‘ (set f)) and disj: snd ‘ set f ∩ Nts(set ps)
= {}
    and A: A ∉ snd ‘ set f
  shows lang (replace_Tm_2 f ps) A = lang ps A
  apply (unfold set_replace_Tm_2[OF dist])
  apply (rule Lang_Replace_Tm_2)
  using assms
  by(simp_all add: dom_map_of_conv_image_fst ran_distinct)

lemma map_replace_Tm_sym_id: α = map (replace_Tm_sym f) α ⟷ Tms_syms
α ∩ dom f = {}
by(induction α)(auto simp: replace_Tm_sym_def split: sym.split)

lemma uniform_Replace_Tm_2:
  assumes Pf: Bad_tms P ⊆ dom f shows uniform (Replace_Tm_2 f P)
  unfolding uniform_def
proof (safe del: disjCI)
  fix A β assume Aβ: (A,β) ∈ Replace_Tm_2 f P
  show (∄t. Tm t ∈ set β) ∨ (∃t. β = [Tm t])
  proof(cases (A,β) ∈ Replace_Tm_new f)
    case True
    then show ?thesis by (auto simp: Replace_Tm_new_def)
  next
    case False
    with Aβ obtain α where Aα: (A,α) ∈ P
      and [simp]: β = (if length α < 2 then α else map (replace_Tm_sym f) α)
     by (auto simp: Replace_Tm_2_def Replace_Tm_def replace_Tm_2_syms_def)
    show ?thesis
    proof (cases length α < 2)
      case True
      then show ?thesis
      by (auto simp: numeral_2_eq_2 less_Suc_eq_le le_Suc_eq length_Suc_conv
            replace_Tm_sym_def)
    next
      case False
      { fix a assume Tm a ∈ set α
        with False Aα have a ∈ Bad_tms P
          by (auto simp: Bad_tms_def Tms_syms_def split: prod.splits)
        with Pf have a ∈ dom f by auto
      } note ∗ = this
      show ?thesis
        by (auto simp: False replace_Tm_sym_def dest!: ∗ split: sym.splits)
    qed
  qed
qed

definition Uniformize :: ('n::fresh0) ⇒ 't list ⇒ ('n, 't) Prods ⇒ ('n, 't) Prods
where
```

[*code_unfold*]: *Uniformize S ts P = Replace_Tm_2 (fresh_map (insert S (Nts P)) ts) P*

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

definition *uniformize* :: *('n::fresh0) ⇒ ('n, 't) prods ⇒ ('n, 't) prods* **where**
uniformize S ps =
(let ts = bad_tms ps;
tmap = fresh_assoc (insert S (Nts(set ps))) ts
in replace_Tm_2 tmap ps)

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

lemma *distinct_bad_tms*: *distinct (bad_tms ps)*
by (*simp add*: *bad_tms_def*)

lemma *set_uniformize*: *set (uniformize S ps) = Uniformize S (bad_tms ps) (set ps)*
by (*simp add*: *uniformize_def Uniformize_def*
set_replace_Tm_2 map_fst_fresh_assoc distinct_bad_tms map_of_fresh_assoc)

lemma *uniform_Uniformize*: *Bad_tms P ⊆ set ts ⟹ uniform (Uniformize S ts P)*
by (*simp add*: *Uniformize_def uniform_Replace_Tm_2 dom_fresh_map*)

lemma *uniform_uniformize*: *uniform (set (uniformize S ps))*
by (*simp add*: *set_uniformize uniform_Uniformize set_bad_tms*)

lemma *Lang_Uniformize*:
assumes *fin*: *finite (Nts P)*
shows *A ∈ Nts P ∪ {S} ⟹ Lang (Uniformize S ts P) A = Lang P A*
apply (*unfold Uniformize_def*)
apply (*subst Lang_Replace_Tm_2*)
using *fresh_map_disj*[*of insert S (Nts P) ts, simplified, OF fin*]
by (*auto simp*: *dom_fresh_map fresh_map_inj_on fin*)

lemma *lang_uniformize*: *A ∈ Nts (set ps) ∪ {S} ⟹ lang (uniformize S ps) A = lang ps A*
by (*auto simp*: *set_uniformize Lang_Uniformize finite_Nts*)

lemma *Eps_free_Uniformize*: *Eps_free P ⟹ Eps_free (Uniformize S ts P)*
by (*auto simp*: *Eps_free_def Uniformize_def*
Replace_Tm_2_def Replace_Tm_def replace_Tm_2_syms_def Replace_Tm_new_def)

lemma *eps_free_uniformize*: *eps_free ps ⟹ eps_free (uniformize S ps)*

by (*simp add*: *set_uniformize Eps_free_Uniformize*)

lemma *Unit_free_Uniformize*: *Unit_free P* $\Longrightarrow$ *Unit_free* (*Uniformize S ts P*)
apply (*unfold Unit_free_def*)
by (*auto simp add*: *Uniformize_def Replace_Tm_2_def Replace_Tm_def Replace_Tm_new_def replace_Tm_2_syms_def*)

lemma *Unit_free_uniformize*: *Unit_free* (*set ps*) $\Longrightarrow$ *Unit_free* (*set* (*uniformize S ps*))
by (*simp add*: *set_uniformize Unit_free_Uniformize*)

The following is used to prove that binarization preserves uniformity. The latter is characterized in terms of *badTmsCount = 0*.

lemma *Nts_correct*: $A \notin$ *Nts P* $\Longrightarrow$ ($\nexists S\ \alpha.\ (S, \alpha) \in P \wedge (Nt\ A \in \{Nt\ S\} \cup set\ \alpha)$)
unfolding *Nts_def Nts_syms_def* **by** *auto*

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 P = *sum prodTms P*

lemma *uniform_badTmsCount*: **assumes** *finite P*
shows *uniform P* $\longleftrightarrow$ *badTmsCount P* = *0*
proof
assume *assm*: *uniform P*
have $\forall p \in P.$ *prodTms p* = *0*
proof
fix *p* **assume** $p \in P$
hence ($\nexists t.$ *Tm t* $\in$ *set* (*snd p*)) $\vee$ ($\exists t.$ *snd p* = [*Tm t*])
using *assm* **unfolding** *uniform_def* **by** *auto*
hence *length* (*snd p*) $\leq$ *1* $\vee$ ($\nexists t.$ *Tm t* $\in$ *set* (*snd p*))
by *auto*
hence *length* (*snd p*) $\leq$ *1* $\vee$ *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 P* = *0*
using *assms* **by** *auto*
next
assume *assm*: *badTmsCount P* = *0*
have $\forall p \in P.$ (($\nexists t.$ *Tm t* $\in$ *set* (*snd p*)) $\vee$ ($\exists t.$ *snd p* = [*Tm t*]))
proof
fix *p* **assume** $p \in P$

```
    hence prodTms p = 0
      using assm assms by auto
    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 P
    unfolding uniform_def by auto
qed
```

10.2 Binarization

Binarization has two parts: a relational specification of what a single step in the conversion should do and an executable function that performs the transitive-reflexive closure of a single step. This way multiple functional implementations can be proved correct more easily. The relational part is inherited from Aditi Barthwal's work.

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

fun badNtsCount :: ('n,'t) Prods ⇒ nat where
  badNtsCount P = sum prodNts P

lemma badNtsCountSet: assumes finite P
  shows (∀p ∈ P. prodNts p = 0) ⟷ badNtsCount P = 0
  using assms by simp

lemma binary_badNtsCount:
  assumes finite P uniform P badNtsCount P = 0
  shows binary P
proof −
  have ∀p ∈ P. length (snd p) ≤ 2
  proof
    fix p assume assm: p ∈ P
    obtain A α where (A, α) = p
      using prod.collapse by blast
    hence ((∄t. Tm t ∈ set α) ∨ (∃t. α = [Tm t])) ∧ (prodNts (A, α) = 0)
      using assms badNtsCountSet assm unfolding uniform_def by auto
   hence ((∄t. Tm t ∈ set α) ∨ (∃t. α = [Tm t])) ∧ (length α ≤ 2 ∨ length (filter
(isNt) α) = 0)
      unfolding prodNts_def by force
    hence ((∄t. Tm t ∈ set α) ∨ (length α ≤ 1)) ∧ (length α ≤ 2 ∨ (∄N. Nt N
∈ set α))
```

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*) ≤ 2
using ‹(*A*, α) = *p*› **by** *auto*
qed
thus *?thesis*
by (*auto simp*: *binary_def*)
qed

10.2.1 Specification of a Single Binarization Step

definition *binarizeStep* :: 'n::*infinite* ⇒ 'n ⇒ 'n ⇒ 'n ⇒ ('n,'t)*Prods* ⇒ ('n,'t)*Prods* ⇒ *bool* **where**
binarizeStep A B_1 B_2 *S P P'* ≡ (
$\exists$ *l r p s*. (*l*,*r*) ∈ *P* ∧ (*r* = *p*@[*Nt* B_1,*Nt* B_2]@*s*)
∧ (*p* ≠ [] ∨ *s* ≠ []) ∧ (*A* ∉ (*Nts P* ∪ {*S*}))
∧ *P'* = *P* − {(*l*,*r*)} ∪ {(*A*, [*Nt* B_1,*Nt* B_2]), (*l*, *p*@[*Nt A*]@*s*)})

lemma *binarizeStep_Eps_free*:
assumes *Eps_free P*
and *binarizeStep A* B_1 B_2 *S P P'*
shows *Eps_free P'*
using *assms* **unfolding** *binarizeStep_def Eps_free_def* **by** *force*

lemma *binarizeStep_Unit_free*:
assumes *Unit_free P*
and *binarizeStep A* B_1 B_2 *S P P'*
shows *Unit_free P'*
proof −
have *1*: ($\nexists$ *l A*. (*l*,[*Nt A*]) ∈ *P*)
using *assms*(*1*) **unfolding** *Unit_free_def* **by** *simp*
obtain *l r p s* **where** *lrps*: (*l*,*r*) ∈ *P* ∧ (*r* = *p*@[*Nt* B_1,*Nt* B_2]@*s*) ∧ (*p* ≠ [] ∨ *s* ≠ [])
∧ (*P'* = ((*P* − {(*l*,*r*)}) ∪ {(*A*, [*Nt* B_1,*Nt* B_2]), (*l*, *p*@[*Nt A*]@*s*)}))
using *assms*(*2*) **unfolding** *binarizeStep_def* **by** *blast*
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'*]) ∈ ((*P* − {(*l*,*r*)}) ∪ {(*A*, [*Nt* B_1,*Nt* B_2]), (*l*, *p*@[*Nt A*]@*s*)})
using *1* **by** *simp*
moreover have *P'* = ((*P* − {(*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 *cnf_r1Nt*:
assumes *binarizeStep A* B_1 B_2 *S P P'*

and $P \vdash lhs \Rightarrow rhs$
shows $P' \vdash lhs \Rightarrow* rhs$
proof −
obtain $p'\ s'\ C\ v$ **where** Cv: $lhs = p'@[Nt\ C]@s' \wedge rhs = p'@v@s' \wedge (C,v) \in P$
using *derive.cases*[*OF assms*(*2*)] **by** *fastforce*
obtain $l\ r\ p\ s$ **where** *lrps*: $(l,r) \in P \wedge (r = p@[Nt\ B_1,Nt\ B_2]@s) \wedge (p \neq [] \vee s \neq []) \wedge (A \notin Nts\ P)$
$\wedge\ (P' = ((P - \{(l,r)\}) \cup \{(A, [Nt\ B_1,Nt\ B_2]), (l, p@[Nt\ A]@s)\}))$
using *assms*(*1*) *set_removeAll* **unfolding** *binarizeStep_def* **by** *fastforce*
thus *?thesis*
proof (*cases* $(C, v) \in P'$)
case *True*
then show *?thesis*
using *derive.intros*[*of C v*] *Cv* **by** *blast*
next
case *False*
hence $C = l \wedge v = p@[Nt\ B_1,Nt\ B_2]@s$
by (*simp add*: *lrps Cv*)
have *1*: $P' \vdash [Nt\ l] \Rightarrow p@[Nt\ A]@s$
using *lrps* **by** (*simp add*: *derive_singleton*)
have $P' \vdash [Nt\ A] \Rightarrow [Nt\ B_1,Nt\ B_2]$
using *lrps* **by** (*simp add*: *derive_singleton*)
hence $P' \vdash [Nt\ l] \Rightarrow* p@[Nt\ B_1,Nt\ B_2]@s$
by (*meson 1 converse_rtranclp_into_rtranclp derive_append derive_prepend r_into_rtranclp*)
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_1Nt*:
assumes *binarizeStep A* B_1 B_2 *S P P'*
and $(A, \alpha) \in P'$
shows $\alpha = [Nt\ B_1,Nt\ B_2]$
proof −
have $A \notin Nts\ P$
using *assms*(*1*) **unfolding** *binarizeStep_def* **by** *blast*
hence $\nexists\alpha.\ (A, \alpha) \in P$
unfolding *Nts_def* **by** *auto*
hence $\nexists\alpha.\ \alpha \neq [Nt\ B_1,Nt\ B_2] \wedge (A, \alpha) \in P'$
using *assms*(*1*) **unfolding** *binarizeStep_def* **by** *auto*
thus *?thesis*
using *assms*(*2*) **by** *blast*
qed

lemma *slemma4_4Nt*:
assumes *binarizeStep A* B_1 B_2 *S P P'*
and $(l,r) \in P$

shows $(Nt\ A) \notin set\ r$
proof −
have $A \notin Nts\ P$
using *assms(1)* **unfolding** *binarizeStep_def* **by** *blast*
hence $\nexists S\ \alpha.\ (S,\ \alpha) \in P \wedge (Nt\ A \in \{Nt\ S\} \cup set\ \alpha)$
using *Nts_correct*[*of A* ‹*P*›] **by** *blast*
thus *?thesis*
using *assms(2)* **by** *blast*
qed

lemma *lemma1Nt*:
assumes *binarizeStep A* B_1 B_2 *S P P'*
and $P' \vdash lhs \Rightarrow rhs$
shows (*substsNt A* [*Nt* B_1,*Nt* B_2] *lhs* = *substsNt A* [*Nt* B_1,*Nt* B_2] *rhs*)
$\vee$ ($P \vdash$ (*substsNt A* [*Nt* B_1,*Nt* B_2] *lhs*) $\Rightarrow$ *substsNt A* [*Nt* B_1,*Nt* B_2] *rhs*)
proof −
obtain *l r p s* **where** *lrps*: $(l,r) \in P \wedge (r = p@[Nt\ B_1,Nt\ B_2]@s) \wedge (p \neq [] \vee s \neq []) \wedge (A \notin Nts\ P)$
$\wedge\ (P' = ((P - \{(l,r)\}) \cup \{(A,\ [Nt\ B_1,Nt\ B_2]),\ (l,\ p@[Nt\ A]@s)\}))$
using *assms(1) set_removeAll* **unfolding** *binarizeStep_def* **by** *fastforce*
obtain *p' s' u v* **where** *uv*: $lhs = p'@[Nt\ u]@s' \wedge rhs = p'@v@s' \wedge (u,v) \in P'$
using *derive.cases*[*OF assms(2)*] **by** *fastforce*
thus *?thesis*
proof (*cases* $u = A$)
case *True*
then show *?thesis*
proof (*cases* $v = [Nt\ B_1,Nt\ B_2]$)
case *True*
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 A*]@*s'*)
using *uv* ‹$u = A$› **by** *simp*
hence *1*: *substsNt A* [*Nt* B_1,*Nt* B_2] *lhs* = *substsNt A* [*Nt* B_1,*Nt* B_2] *p'* @ [*Nt* B_1,*Nt* B_2] @ *substsNt A* [*Nt* B_1,*Nt* B_2] *s'*
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] ([*Nt* B_1,*Nt* B_2]@*s'*)
using *uv* ‹$u = A$› *True* **by** *simp*
hence *substsNt A* [*Nt* B_1,*Nt* B_2] *rhs* = *substsNt A* [*Nt* B_1,*Nt* B_2] *p'* @ [*Nt* B_1,*Nt* B_2] @ *substsNt A* [*Nt* B_1,*Nt* B_2] *s'*
using *assms(1)* **unfolding** *binarizeStep_def Nts_def* **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 P P'*] **unfolding** *binarizeStep_def Nts_def* **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* **by** (*simp add*: *substs_skip*)
hence *2*: (*u*, *substsNt A* [*Nt B*$_1$,*Nt B*$_2$] *v*) ∈ *P*
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*) ∈ *P*
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 B*$_1$,*Nt B*$_2$] *s'*
using *False substs_skip* **by** *fastforce*

```
      thus ?thesis
        using 1 2 assms(2) uv derive.simps by fast
    qed
  qed
qed

lemma lemma3Nt:
  assumes P' ⊢ lhs ⇒* rhs
    and binarizeStep A B1 B2 S P P'
  shows P ⊢ 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 P P' y z] by auto
qed simp

lemma Lang_binarizeStep1:
  assumes binarizeStep A B1 B2 S P P'
  shows Lang P' S ⊆ Lang P S
proof
  fix w
  assume w ∈ Lang P' S
  hence P' ⊢ [Nt S] ⇒* map Tm w
    by (simp add: Lang_def)
  hence P' ⊢ [Nt S] ⇒* map Tm w
    using assms unfolding binarizeStep_def by auto
  hence P ⊢ substsNt A [Nt B1, Nt B2] [Nt S] ⇒*  substsNt A [Nt B1, Nt B2]
(map Tm w)
    using assms lemma3Nt[of P' ‹[Nt S]› ‹map Tm w›] by blast
  moreover have substsNt A [Nt B1, Nt B2] [Nt S] = [Nt S]
    using assms unfolding binarizeStep_def by auto
  moreover have substsNt A [Nt B1, Nt B2] (map Tm w) = map Tm w by simp
  ultimately show w ∈ Lang P S using Lang_def
    by (metis (no_types, lifting) mem_Collect_eq)
qed

lemma slemma5_1Nt:
  assumes P ⊢ u ⇒* v
    and binarizeStep A B1 B2 S P P'
  shows P' ⊢ u ⇒* v
  using assms by (induction v rule: rtranclp_induct) (auto simp: cnf_r1Nt rtran-
clp_trans)

lemma Lang_binarizeStep2:
  assumes binarizeStep A B1 B2 S P P'
  shows Lang P S ⊆ Lang P' S
proof
  fix w
```

assume $w \in Lang\ P\ S$
hence $P \vdash [Nt\ S] \Rightarrow* map\ Tm\ w$
using *assms* **unfolding** *Lang_def binarizeStep_def* **by** *auto*
thus $w \in Lang\ P'\ S$
using *assms slemma5_1Nt Lang_def* **by** *fast*
qed

lemma *Lang_binarizeStep*: $binarizeStep\ A\ B_1\ B_2\ S\ P\ P' \Longrightarrow Lang\ P\ S = Lang\ P'\ S$
using *Lang_binarizeStep1 Lang_binarizeStep2* **by** *fast*

lemma *Eps_free_binarizeStepRtc*:
assumes $(\lambda x\ y.\ \exists A\ t\ B_1\ B_2.\ binarizeStep\ A\ B_1\ B_2\ S\ x\ y)^{**}\ P\ P'$
and *Eps_free P*
shows *Eps_free P'*
using *assms* **by** (*induction rule*: *rtranclp_induct*) (*auto simp*: *binarizeStep_Eps_free*)

lemma *Unit_free_binarizeStepRtc*:
assumes $(\lambda x\ y.\ \exists A\ t\ B_1\ B_2.\ binarizeStep\ A\ B_1\ B_2\ S\ x\ y)^{**}\ P\ P'$
and *Unit_free P*
shows *Unit_free P'*
using *assms* **by** (*induction rule*: *rtranclp_induct*) (*auto simp*: *binarizeStep_Unit_free*)

theorem *Lang_binarizeStepRtc*:
assumes $(\lambda x\ y.\ \exists A\ B_1\ B_2.\ binarizeStep\ A\ B_1\ B_2\ S\ x\ y)^{**}\ P\ P'$
shows $Lang\ P\ S = Lang\ P'\ S$
using *assms* **by** (*induction rule*: *rtranclp_induct*) (*fastforce simp*: *Lang_binarizeStep*)+

Termination

lemma *lemma6_b*:
assumes *finite P binarizeStep A B*$_1$ *B*$_2$ *S P P'* **shows** *badNtsCount P' < badNtsCount P*
proof −
from *assms*(*2*) **obtain** *l r p s* **where** *lrps*: $(l,r) \in P\ r = p@[Nt\ B_1,Nt\ B_2]@s\ p \neq [] \vee s \neq []$
$A \notin Nts\ P\ P' = P - \{(l,r)\} \cup \{(A, [Nt\ B_1,Nt\ B_2]), (l, p@[Nt\ A]@s)\}$
unfolding *binarizeStep_def* **by** *auto*
let *?B12* $= [Nt\ B_1,Nt\ B_2]::('a,'b)syms$
have *prodNts* $(l,p@?B12@s) = length\ (filter\ isNt\ (p@?B12@s))$
using *lrps* **unfolding** *prodNts_def* **by** *auto*
hence *1*: $prodNts\ (l,p@?B12@s) = length\ (filter\ isNt\ (p@s)) + 2$
by (*simp add*: *isNt_def*)
have $(A,?B12) \notin P \wedge (l,\ p@[Nt\ A]@s) \notin P$
using *Nts_correct*[*OF* ‹$A \notin Nts\ P$›] **by** *fastforce*
then have $badNtsCount\ P' = badNtsCount\ (P - \{(l,r)\}) + badNtsCount\ \{(A,?B12), (l,\ p@[Nt\ A]@s)\}$
unfolding *badTmsCount.simps* ‹$P' = \_$› **by** (*simp add*: *assms*(*1*) *sum_Un_eq*)
also have $\ldots = badNtsCount\ (P - \{(l,r)\}) + badNtsCount\ \{(A,?B12)\} + badNtsCount\{(l,\ p@[Nt\ A]@s)\}$

```
    using Nts_correct[OF ‹A ∉ Nts P›] lrps(1) by simp
  finally have 2: badNtsCount P' = ... .
  have 3: badNtsCount (P − {(l,r)}) < badNtsCount P
    using sum.remove[OF assms(1) lrps(1), of prodNts] lrps(2) 1 by (simp)
  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 P' = badNtsCount (P − {(l,r)}) + 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
     by(simp add: sum.remove[OF assms(1) lrps(1)] ‹r = _›)
  next
    assume prodNts (l, p@[Nt A]@s) = 0
    hence badNtsCount P' = badNtsCount (P − {(l,r)})
      using 2 by (simp add: prodNts_def)
    thus ?thesis
      using 3 by simp
  qed
qed

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 badNtsCountNot0:
  assumes finite P badNtsCount P > 0
  shows ∃ l r. (l, r) ∈ P ∧ length r ≥ 3
using assms badNtsCountSet not_gr0 unfolding prodNts_def by fastforce
```

10.2.2 Functional Binarization

```
definition freshA :: ('n::fresh0,'t) prods ⇒ 'n ⇒ 'n where
  freshA ps S = fresh0 (Nts (set ps) ∪ {S})

lemma freshA_notin_set:
  shows freshA ps S ∉ (Nts (set ps) ∪ {S})
```

unfolding *freshA_def* **by** (*metis ID.set_finite finite_Un finite_nts fresh0_notIn*)

fun *replaceNts* :: *'n::fresh0* ⇒ (*'n*,*'t*) *syms* ⇒ (*'n* × *'n*) *option* × (*'n*,*'t*) *syms*
where
replaceNts A [] = (*None*, []) |
replaceNts A [*s*] = (*None*, [*s*]) |
replaceNts A (*Nt* s_1 # *Nt* s_2 # *sl*) = (*Some* (s_1, s_2), *Nt A* # *sl*) |
replaceNts A (*s*#*sl*) = (*let* (*nn_opt*, *sl'*) = *replaceNts A sl in* (*nn_opt*, *s*#*sl'*))

lemma *replaceNts_tm_unchanged_opt*:
assumes
replaceNts A (*s0*#*s1*#*sl*) = (*nn_opt*, *sl'*)
∃ *t*. *s0* = *Tm t* ∨ *s1* = *Tm t*
obtains *sl''* **where** *replaceNts A* (*s1*#*sl*) = (*nn_opt*, *sl''*)
proof −
obtain *nn_opt' sl''* **where** *replaceNts A* (*s1*#*sl*) = (*nn_opt'*, *sl''*)
by *fastforce*
moreover with *assms* **have** *nn_opt* = *nn_opt'* **by** *fastforce*
ultimately show *thesis* **using** *that* **by** *blast*
qed

lemma *replaceNts_id_iff_None*:
assumes *replaceNts A sl* = (*nn_opt*, *sl'*)
shows *nn_opt* = *None* ⟷ *sl* = *sl'*
using *assms* **proof** (*induction sl arbitrary*: *nn_opt sl' rule*: *replaceNts.induct*)
case (*4_1 A t s sl*)
then obtain *sl''* **where** *rec*: *replaceNts A* (*s*#*sl*) = (*nn_opt*, *sl''*)
using *replaceNts_tm_unchanged_opt* **by** *blast*
then show *?case* **using** *4_1* **by** *auto*
next
case (*4_2 A s t sl*)
then obtain *sl''* **where** *rec*: *replaceNts A* (*Tm t*#*sl*) = (*nn_opt*, *sl''*)
using *replaceNts_tm_unchanged_opt* **by** *blast*
then show *?case* **using** *4_2* **by** *auto*
qed *auto*

lemma *replaceNts_replaces_pair*:
assumes
replaceNts A sl = (*nn_opt*, *sl'*)
nn_opt ≠ *None*
obtains *p q* B_1 B_2 **where**
nn_opt = *Some* (B_1,B_2)
sl = *p*@[*Nt* B_1, *Nt* B_2]@*q*
sl' = *p*@[*Nt A*]@*q*
using *assms* **proof** (*induction sl arbitrary*: *thesis nn_opt sl' rule*: *replaceNts.induct*)
case (*4_1 A t s sl*)
then obtain *sl''* **where**
replaceNts A (*s*#*sl*) = (*nn_opt*, *sl''*)

```
    and sl'_def: sl' = Tm t # sl''
    using replaceNts_tm_unchanged_opt
    by (metis (lifting) case_prod_conv prod.inject replaceNts.simps(4))
  with 4_1(1,4) obtain p q B1 B2 where
    nn_opt = Some (B1,B2) s#sl = p@[Nt B1,Nt B2]@q sl'' = p@[Nt A]@q
    by blast
  moreover with sl'_def have Tm t #s#sl = (Tm t#p)@[Nt B1,Nt B2]@q sl' =
(Tm t#p)@[Nt A]@q
    by auto
  ultimately show ?case using 4_1(2) by blast
next
  case (4_2 A s t sl)
  then obtain sl'' where
    replaceNts A (Tm t#sl) = (nn_opt, sl'')
    and sl'_def: sl' = s # sl''
    using replaceNts_tm_unchanged_opt
    by (metis (lifting) old.prod.case prod.inject replaceNts.simps(5))
  with 4_2(1,4) obtain p q B1 B2 where
    nn_opt = Some (B1,B2) Tm t#sl = p@[Nt B1,Nt B2]@q sl'' = p@[Nt A]@q
    by blast
   moreover with sl'_def have s#Tm t#sl = (s#p)@[Nt B1,Nt B2]@q sl' =
(s#p)@[Nt A]@q
    by auto
  ultimately show ?case using 4_2(2) by blast
qed fastforce+

corollary replaceNts_replaces_pair_Some:
  assumes replaceNts A sl = (Some (B1,B2), sl')
  obtains p q where
    sl = p@[Nt B1, Nt B2]@q
    sl' = p@[Nt A]@q
  using replaceNts_replaces_pair
  by (smt (verit) assms option.distinct(1) option.inject prod.inject)

fun binarize1 :: 'n::fresh0 ⇒ ('n,'t) prods ⇒ ('n,'t) prods ⇒ ('n,'t) prods where
  binarize1 A ps0 [] = ps0 |
  binarize1 A ps0 ((l,r)#ps) =
    (case replaceNts A r of
      (None, _) ⇒ binarize1 A ps0 ps|
      (Some (B1,B2), r') ⇒
        if length r < 3 then binarize1 A ps0 ps
        else (A, [Nt B1,Nt B2]) # (l, r') # removeAll (l,r) ps0)

lemma binarize1_rec_if_id_or_lt3:
  assumes
    replaceNts A r = (nn_opt, r')
    r = r' ∨ length r < 3
  shows binarize1 A ps0 ((l,r)#ps) = binarize1 A ps0 ps
  using assms replaceNts_id_iff_None by (cases nn_opt) auto
```

lemma *binarize1_binarizes*:
assumes *binarize1 A ps0 ps* $\neq$ *ps0*
obtains *l r r′* B_1 B_2 **where**
(*l*,*r*) $\in$ *set ps*
length r > *2*
replaceNts A r = (*Some* (B_1,B_2), *r′*)
binarize1 A ps0 ps = (*A*, [*Nt* B_1,*Nt* B_2]) # (*l*, *r′*) # *removeAll* (*l*,*r*) *ps0*
using *assms* **proof** (*induction ps arbitrary*: *thesis*)
case (*Cons p ps*)
obtain *l r r′ nn_opt* **where** *lr_defs*: *p* = (*l*,*r*) *replaceNts A r* = (*nn_opt*,*r′*)
by *fastforce*
consider (*hd*) *r* $\neq$ *r′* $\wedge$ *length r* > *2* | (*tl*) *r* = *r′* $\vee$ *length r* < *3* **by** *fastforce*
then show *?case*
proof *cases*
case *hd*
with *replaceNts_id_iff_None lr_defs* **obtain** B_1 B_2 **where** *nn_opt* = *Some* (B_1,B_2)
by *fast*
moreover from *this hd* **have**
binarize1 A ps0 (*p*#*ps*) = (*A*, [*Nt* B_1,*Nt* B_2]) # (*l*, *r′*) # *removeAll* (*l*,*r*) *ps0*
using *lr_defs* **by** *auto*
ultimately show *?thesis* **using** *Cons*(*2*) *lr_defs hd* **by** *fastforce*
next
case *tl*
with *lr_defs binarize1_rec_if_id_or_lt3*
have *binarize1 A ps0* (*p*#*ps*) = *binarize1 A ps0 ps* **by** *blast*
with *Cons* **show** *?thesis* **using** *lr_defs* **by** (*metis list.set_intros*(*2*))
qed
qed *simp*

lemma *binarizeStep_binarize1*:
assumes
A $\notin$ *Nts* (*set ps*) $\cup$ {*S*}
binarize1 A ps ps $\neq$ *ps*
obtains B_1 B_2 **where** *binarizeStep A* B_1 B_2 *S* (*set ps*) (*set* (*binarize1 A ps ps*))
proof −
from *binarize1_binarizes*[*OF assms*(*2*)] **obtain** *l r r′* B_1 B_2 **where**
binarize1_defs:
(*l*,*r*) $\in$ *set ps*
length r > *2*
replaceNts A r = (*Some* (B_1,B_2), *r′*)
binarize1 A ps ps = (*A*, [*Nt* B_1,*Nt* B_2]) # (*l*, *r′*) # *removeAll* (*l*,*r*) *ps*
by *metis*
moreover from *this* **obtain** *p s* **where** *r* = *p*@[*Nt* B_1, *Nt* B_2]@*s* *r′* = *p*@[*Nt A*]@*s*
using *replaceNts_replaces_pair* **by** *blast*

```
  ultimately have binarizeStep A B₁ B₂ S (set ps) (set (binarize1 A ps ps))
    unfolding binarizeStep_def using assms(1) by auto
  then show thesis using that by simp
qed

lemma binarize1_dec_badNtsCount:
  assumes binarize1 A ps ps ≠ ps
          A ∉ Nts (set ps) ∪ {S}
  shows badNtsCount (set (binarize1 A ps ps)) < badNtsCount (set ps)
  using lemma6_b assms binarizeStep_binarize1
  by (metis list.set_finite)

lemma badNts_impl_binarize1_not_id_unif:
  assumes badNtsCount (set ps) = Suc n
    uniform (set ps)
  shows binarize1 A ps0 ps ≠ ps0
  using assms proof (induction ps arbitrary: n)
  case (Cons p ps)
  obtain l r where lr_def: (l,r) = p using old.prod.exhaust by metis
  consider (no_badNts_hd) badNtsCount (set [p]) = 0 |
           (Suc_badNts_hd) m where badNtsCount (set [p]) = Suc m
    using not0_implies_Suc by blast
  then show ?case
  proof cases
    case no_badNts_hd
    from Cons(3) have only_Nts: length r = 1 ∨ (∀ s∈(set r). ∃ n. s = Nt n)
      unfolding uniform_def using lr_def
       by (smt (verit, best) One_nat_def case_prodD destTm.cases length_Cons
list.set_intros(1)
          list.size(3))
    have length r < 3
    proof (rule ccontr)
      assume ¬?thesis
      hence length r ≥ 2 by simp
      moreover with only_Nts have ∀ s∈set r. ∃ n. s = Nt n by presburger
      ultimately have prodNts p ≠ 0 unfolding prodNts_def using lr_def
     by (metis ‹¬ length r < 3› filter_True isNt_def le_imp_less_Suc not_numeral_le_zero
numeral_2_eq_2
           numeral_3_eq_3 snd_conv)
     with no_badNts_hd show False by simp
    qed
    with lr_def have binarize1 A ps0 (p#ps) = binarize1 A ps0 ps
      using binarize1_rec_if_id_or_lt3 by (metis old.prod.exhaust)
    with Cons show ?thesis
      by (metis (no_types, lifting) badNtsCountSet bot_nat_0.not_eq_extremum
gr0_conv_Suc
       list.set_finite list.set_intros(1,2) no_badNts_hd set_ConsD uniform_def)
  next
    case Suc_badNts_hd
```

```
    with lr_def have all_Nts: length r > 2 ∧ (∀ s∈set r. ∃ n. s = Nt n)
    using Cons.prems(2) uniform_badTmsCount[of set (p # ps)] noTms_prodTms0[of
l r]
     by(auto simp: prodNts_def length_Suc_conv isNt_def split: if_splits)
    moreover obtain r' B1 B2 where replace_defs: replaceNts A r = (Some
(B1,B2), r') r' ≠ r
    proof −
     from all_Nts obtain ns B1 B2 where r = [Nt B1, Nt B2] @ ns
       by (metis (no_types, lifting) append_Cons append_Nil le_imp_less_Suc
length_Suc_conv
       linorder_not_less list.exhaust list.set_intros(1,2) list.size(3) not_less_iff_gr_or_eq

          numeral_2_eq_2)
     thus thesis using that by simp
    qed
    ultimately have binarize1 A ps0 (p#ps) = (A, [Nt B1, Nt B2]) # (l,r') #
removeAll (l,r) ps0
                  (is _ = ?rem)
     using lr_def by fastforce
    also have ... ≠ ps0
    proof
     assume rem_eq: ... = ps0
     then obtain xs where ps0 = (A, [Nt B1, Nt B2]) # (l,r') # xs by metis
     with rem_eq have (l,r) = (l,r') using set_removeAll
      by (smt (verit, ccfv_SIG) Diff_disjoint insert_disjoint(2) length_Cons lessI
not_add_less2
          plus_1_eq_Suc removeAll.simps(2) removeAll_id)
     with replace_defs show False by blast
    qed
    finally show ?thesis .
  qed
qed simp


lemma uniform_binarize1:
  fixes ps :: ('n::fresh0, 't) prods
  assumes ps_uniform: uniform (set ps)
    shows uniform (set( binarize1 A ps ps))
proof −
  consider (id) binarize1 A ps ps = ps | (not_id) binarize1 A ps ps ≠ ps by blast
  then show ?thesis
  proof cases
    case not_id
    from binarize1_binarizes[OF not_id] obtain l r r' B1 B2 where lr_defs:
      (l,r) ∈ set ps length r > 2 replaceNts A r = (Some (B1,B2), r')
     binarize1 A ps ps = (A,[Nt B1, Nt B2]) # (l,r') # removeAll (l,r) ps by metis
    moreover from ps_uniform have uniform (set (removeAll (l,r) ps))
      unfolding uniform_def by simp
    moreover have uniform (set [(l,r')])
```

```
    proof −
      from replaceNts_replaces_pair_Some[OF lr_defs(3)] obtain p q where
        r = p@[Nt B1,Nt B2]@q r' = p@[Nt A]@q .
      with lr_defs ps_uniform show ?thesis unfolding uniform_def by fastforce
    qed
    ultimately show ?thesis unfolding uniform_def by auto
  qed (use assms in simp)
qed

function binarize :: 'n::fresh0 ⇒ ('n,'t) prods ⇒ ('n,'t) prods where
  binarize S ps =
    (let ps' = binarize1 (freshA ps S) ps ps in
    if ps = ps' then ps else binarize S ps')
  by auto
termination
proof
  let ?R = measure (λ(S,ps). badNtsCount (set ps))
  show wf ?R by fast
  fix S :: 'n::fresh0
  and ps ps' :: ('n,'t) prods
  let ?A = freshA ps S
  assume ps'_def: ps' = binarize1 ?A ps ps
        and neq: ps ≠ ps'
  moreover with freshA_notin_set have ?A ∉ Nts (set ps) ∪ {S} by blast
  ultimately show ((S,ps'),(S,ps)) ∈ ?R
    using binarize1_dec_badNtsCount by force
qed

lemma binarize_binarizeStep:
  (λx y. ∃A B1 B2. binarizeStep A B1 B2 S x y)** (set ps) (set (binarize S ps))
proof (induction badNtsCount (set ps) arbitrary: ps rule: less_induct)
  case less
  let ?A = freshA ps S
  have A_notin_nts: ?A ∉ Nts (set ps) ∪ {S}
    using freshA_notin_set by metis
  consider (eq) binarize1 ?A ps ps = ps |
           (neq) binarize1 ?A ps ps ≠ ps by blast
  then show ?case
  proof cases
    case neq
    let ?ps' = binarize1 ?A ps ps
    from binarize1_dec_badNtsCount[OF neq A_notin_nts] have
      badNtsCount (set ?ps') < badNtsCount (set ps) .
    with less have (λx y. ∃A B1 B2. binarizeStep A B1 B2 S x y)** (set ?ps') (set
(binarize S ?ps'))
      by blast
    moreover from neq A_notin_nts obtain B1 B2 where binarizeStep ?A B1
B2 S (set ps) (set ?ps')
      using binarizeStep_binarize1 by blast
```

```
    ultimately show ?thesis
      by (smt (verit, best) binarize.simps
          converse_rtranclp_into_rtranclp)
  qed simp
qed

lemma uniform_binarize:
  fixes ps :: ('n::fresh0, 't) prods
  assumes ps_uniform: uniform (set ps)
    shows uniform (set (binarize S ps))
using assms proof (induction badNtsCount (set ps) arbitrary: ps rule: less_induct)
  case less
  let ?A = freshA ps S
  consider (rec) binarize1 ?A ps ps ≠ ps | (no_rec) binarize1 ?A ps ps = ps by
blast
  then show ?case
  proof cases
    case rec
    let ?ps' = binarize1 ?A ps ps
    from rec have binarize S ps = binarize S ?ps'
      by (smt (verit) binarize.elims)
    with less binarize1_dec_badNtsCount[OF rec] freshA_notin_set
      uniform_binarize1
      show ?thesis by metis
  qed (use less in simp)
qed

lemma binary_binarize:
  assumes binary: binary (set ps)
    shows binary (set (binarize S ps))
proof −
  from binary have badNtsCount (set ps) = 0
   by (metis badNtsCountNot0 binary_def bot_nat_0.not_eq_extremum leD le_imp_less_Suc
numeral_2_eq_2
        numeral_3_eq_3 split_conv list.set_finite)
  hence binarize S ps = ps using binarize1_dec_badNtsCount freshA_notin_set
    by (smt (verit, best) binarize.simps bot_nat_0.extremum_strict)
  with assms show ?thesis by argo
qed

lemma binarize_binary_if_uniform:
  fixes ps :: ('n::fresh0, 't) prods
  assumes uniform: uniform (set ps)
    shows binary (set (binarize S ps))
proof −
  consider (bin) binary (set ps) | (not_bin) ¬binary (set ps) by blast
  then show ?thesis
  proof cases
    case bin
```

```
    then show ?thesis using binary_binarize by blast
  next
    case not_bin
   with uniform binary_badNtsCount obtain n where Suc_badNts: badNtsCount
(set ps) = Suc n
      using not0_implies_Suc by blast
    with uniform show ?thesis
    proof (induction badNtsCount (set ps) arbitrary: ps n rule: less_induct)
      case less
      let ?A = freshA ps S
      from less badNts_impl_binarize1_not_id_unif have binarize1 ?A ps ps ≠
ps
        by fastforce
       hence badNtsCount_dec: badNtsCount (set (binarize1 ?A ps ps)) < bad-
NtsCount (set ps)
                          (is badNtsCount ?ps' < _)
        using freshA_notin_set binarize1_dec_badNtsCount by metis
       consider (zero_badNts) badNtsCount ?ps' = 0 | (Suc_badNts) m where
badNtsCount ?ps' = Suc m
        using not0_implies_Suc by blast
      then show ?case
      proof cases
        case zero_badNts
        moreover from less.prems(1) uniform_binarize1 have uniform ?ps'
          by blast
        ultimately show ?thesis using binary_badNtsCount
       by (smt (verit, ccfv_threshold) List.finite_set binarize.elims binary_binarize
             freshA_def less.prems(2))
      next
        case Suc_badNts
        moreover from less.prems(1) uniform_binarize1 have unif: uniform ?ps'
          by blast
      ultimately show ?thesis using less(1)[OF badNtsCount_dec _ Suc_badNts]

          by (smt (verit, best) binarize.simps freshA_def less.prems(2))
      qed
    qed
  qed
qed
```

10.3 Conversion to CNF

Alternative form more similar to the one Jana Hofmann used:

```
lemma CNF_eq: CNF P ⟷ (uniform P ∧ binary P ∧ Eps_free P ∧ 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

definition cnf_of :: ('n::fresh0, 't) prods ⇒ 'n ⇒ ('n,'t) prods where
 cnf_of ps S = (binarize S ∘ uniformize S ∘ unit_elim ∘ eps_elim) ps

theorem cnf_of_CNF_Lang:
 fixes ps :: ('n::fresh0, 't) prods
 shows CNF (set(cnf_of ps S)) lang (cnf_of ps S) S = lang ps S − {[]}
proof −
 let ?ps1 = eps_elim ps let ?ps2 = unit_elim ?ps1
 let ?ps3 = uniformize S ?ps2 let ?ps4 = binarize S ?ps3
 have eps_free ?ps1 by (rule eps_free_eps_elim)
 hence eps_free ?ps2 by (meson unit_elim_correct Unit_elim_rel_Eps_free)
 have Unit_free(set ?ps2) by (metis unit_elim_correct Unit_free_if_Unit_elim_rel)
 have eps_free ?ps3 by(rule eps_free_uniformize[OF ‹eps_free ?ps2›])
 have Unit_free(set ?ps3) by (rule Unit_free_uniformize[OF ‹Unit_free(set ?ps2)›])
 have uniform (set ?ps3) by (rule uniform_uniformize)

 have eps_free ?ps4
  using binarize_binarizeStep Eps_free_binarizeStepRtc[OF _ ‹eps_free ?ps3›]
by meson
 moreover have Unit_free(set ?ps4)
  using binarize_binarizeStep Unit_free_binarizeStepRtc[OF _ ‹Unit_free(set
?ps3)›] by meson
 moreover have uniform (set ?ps4)
  by(rule uniform_binarize[OF ‹uniform (set ?ps3)›])
 moreover have binary (set ?ps4)
  by (rule binarize_binary_if_uniform[OF ‹uniform (set ?ps3)›])
 ultimately show CNF (set(cnf_of ps S)) unfolding CNF_eq cnf_of_def
  by(simp only: Let_def comp_def)

 have lang ?ps4 S = lang ?ps3 S using Lang_binarizeStepRtc[OF binarize_binarizeStep,
symmetric] .
 also have ... = lang ?ps2 S by (simp add: lang_uniformize)
 also have ... = lang ?ps1 S by (rule lang_unit_elim)
 also have ... = lang ps S − {[]} by (rule lang_eps_elim)
 finally show lang (cnf_of ps S) S = lang ps S − {[]}
  by (simp add: cnf_of_def)
qed

corollary cnf_exists:
 fixes P :: ('n::fresh0,'t) Prods
 assumes finite P
 shows ∃ P'::('n,'t)Prods. finite P' ∧ CNF P' ∧ Lang P' S = Lang P S − {[]}
proof −
```

obtain *ps* **where** $P = set\ ps$ **by** (*metis finite_list*[*OF assms*])
with *cnf_of_CNF_Lang*[*of ps*] **show** *?thesis* **by** *blast*
qed

Some helpful properties:

lemma *cnf_length_derive*:
assumes $CNF\ P\ P \vdash [Nt\ S] \Rightarrow* \alpha$
shows $length\ \alpha \geq 1$
using *assms CNF_eq Eps_free_derives_Nil length_greater_0_conv less_eq_Suc_le*
by *auto*

lemma *cnf_length_derive2*:
assumes $CNF\ P\ P \vdash [Nt\ A, Nt\ B] \Rightarrow* \alpha$
shows $length\ \alpha \geq 2$
proof −
obtain *u v* **where** *uv*: $P \vdash [Nt\ A] \Rightarrow* u \wedge P \vdash [Nt\ B] \Rightarrow* v \wedge \alpha = u\ @\ v$
using *assms*(*2*) *derives_append_decomp*[*of P* ‹$[Nt\ A]$› ‹$[Nt\ B]$› α] **by** *auto*
hence $length\ u \geq 1 \wedge length\ v \geq 1$
using *cnf_length_derive*[*OF assms*(*1*)] **by** *blast*
thus *?thesis*
using *uv* **by** *simp*
qed

lemma *cnf_single_derive*:
assumes $CNF\ P\ P \vdash [Nt\ S] \Rightarrow* [Tm\ t]$
shows $(S, [Tm\ t]) \in P$
proof −
obtain α **where** α: $P \vdash [Nt\ S] \Rightarrow \alpha \wedge P \vdash \alpha \Rightarrow* [Tm\ t]$
using *converse_rtranclpE*[*OF assms*(*2*)] **by** *auto*
hence *1*: $(S, \alpha) \in P$
by (*simp add*: *derive_singleton*)
have $\nexists A\ B.\ \alpha = [Nt\ A, Nt\ B]$
proof (*rule ccontr*)
assume $\neg\ (\nexists A\ B.\ \alpha = [Nt\ A, Nt\ B])$
from *this* **obtain** *A B* **where** *AB*: $\alpha = [Nt\ A, Nt\ B]$
by *blast*
have $\forall w.\ P \vdash [Nt\ A, Nt\ B] \Rightarrow* w \longrightarrow length\ w \geq 2$
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** $\alpha = [Tm\ a]$
using *1 assms*(*1*) **unfolding** *CNF_def* **by** *auto*
hence $t = a$
using α **by** (*simp add*: *derives_Tm_Cons*)
thus *?thesis*
using *1* ‹$\alpha = [Tm\ a]$› **by** *blast*

qed

lemma *cnf_word*:
assumes *CNF P* $P \vdash [Nt\ S] \Rightarrow* map\ Tm\ w$
and *length w* ≥ 2
shows $\exists A\ B\ u\ v.\ (S, [Nt\ A, Nt\ B]) \in P \wedge P \vdash [Nt\ A] \Rightarrow* map\ Tm\ u \wedge P \vdash [Nt\ B] \Rightarrow* map\ Tm\ v \wedge u@v = w \wedge u \neq [] \wedge v \neq []$
proof −
have *1*: $(S, map\ Tm\ w) \notin P$
using *assms(1) assms(3)* **unfolding** *CNF_def* **by** *auto*
have $\exists \alpha.\ P \vdash [Nt\ S] \Rightarrow \alpha \wedge P \vdash \alpha \Rightarrow* map\ Tm\ w$
using *converse_rtranclpE*[*OF assms(2)*] **by** *auto*
from *this* **obtain** α **where** α: $(S, \alpha) \in P \wedge P \vdash \alpha \Rightarrow* map\ Tm\ w$
by (*auto simp*: *derive_singleton*)
hence $(\nexists t.\ \alpha = [Tm\ t])$
using *1 derives_Tm_Cons*[*of P*] *derives_from_empty* **by** *auto*
hence $\exists A\ B.\ P \vdash [Nt\ S] \Rightarrow [Nt\ A, Nt\ B] \wedge P \vdash [Nt\ A, Nt\ B] \Rightarrow* map\ Tm\ w$
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

11 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* ∧ *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* ⊢ *S* ⇒*cnf w CNF P*
shows *P* ⊢ [*Nt S*] ⇒∗ *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* ⊢ [*Nt A*] ⇒ [*Nt B*, *Nt C*]
by (*simp add*: *derive_singleton*)
moreover have *P* ⊢ [*Nt B*] ⇒∗ *map Tm w* ∧ *P* ⊢ [*Nt C*] ⇒∗ *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* ⟹ (*P* ⊢ [*Nt S*] ⇒∗ *map Tm w* ⟷ *P* ⊢ *S* ⇒*cnf w*)
using *cnf_der*[*of P S w*] *der_cnf*[*of P S w*] **by** *blast*

lemma *path_if_derives*:
assumes *P* ⊢ [*Nt S*] ⇒∗ *map Tm w CNF P*
shows ∃*p*. *P* ⊢ *S* ⇒⟨*p*⟩ *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* ⊢ *S* ⇒⟨*p*⟩ *w CNF P*
shows *P* ⊢ [*Nt S*] ⇒∗ *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* ⇒ ′*n* ⇒ ′*n list* ⇒ ′*t list* ⇒ *bool*
((*2*_ ⊢/ (_/ ⇒⟪_⟫/ _)) [*50*, *0*, *0*, *50*] *50*) **where**
terminal: (*A*, [*Tm a*]) ∈ *P* ⟹ *P* ⊢ *A* ⇒⟪[*A*]⟫ [*a*] |
nonTerminals: ⟦(*A*, [*Nt B*, *Nt C*]) ∈ *P* ∧ (*P* ⊢ *B* ⇒⟪*p*⟫ *w*) ∧ (*P* ⊢ *C* ⇒⟪*q*⟫ *v*)⟧
⟹
P ⊢ *A* ⇒⟪*A*#(*if length p* ≥ *length q then p else q*)⟫ (*w*@*v*)

lemma *path_lpath*: $P \vdash S \Rightarrow\langle p\rangle\ w \Longrightarrow \exists\, p'.\ (P \vdash S \Rightarrow\langle\!\langle p'\rangle\!\rangle\ w) \wedge \mathit{length}\ p' \geq \mathit{length}\ p$
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*: $(P \vdash S \Rightarrow\langle\!\langle p\rangle\!\rangle\ w) \Longrightarrow \mathit{length}\ w \leq 2\hat{}\mathit{length}\ p$
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 ≥ length q*)
case *True*
hence *length v ≤ 2^length p*
using *nonTerminals order_subst1*[*of* ‹*length v*› ‹*λx. 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*› ‹*λx. 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 $\exists\, B\ C\ p'\ a\ b.\ (A, [Nt\ B, Nt\ C]) \in P \wedge w = a@b \wedge$
$((P \vdash B \Rightarrow\langle\!\langle p\rangle\!\rangle\ a \wedge P \vdash C \Rightarrow\langle\!\langle p'\rangle\!\rangle\ b \wedge \mathit{length}\ p \geq \mathit{length}\ p') \vee$
$(P \vdash B \Rightarrow\langle\!\langle p'\rangle\!\rangle\ a \wedge P \vdash C \Rightarrow\langle\!\langle p\rangle\!\rangle\ b \wedge \mathit{length}\ p \geq \mathit{length}\ p'))$
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*: $P \vdash A \Rightarrow\langle p\rangle\ w \Longrightarrow \mathit{length}\ w > 0$
by (*induction rule*: *path.induct*) *simp_all*

lemma *substitution*:
assumes $P \vdash A \Rightarrow\langle p1@[X]@p2\rangle\ z$
shows $\exists\, v\ w\ x.\ P \vdash X \Rightarrow\langle [X]@p2\rangle\ w \wedge z = v@w@x \wedge$
$(\forall\, w'\ p'.\ P \vdash X \Rightarrow\langle [X]@p'\rangle\ w' \longrightarrow P \vdash A \Rightarrow\langle p1@[X]@p'\rangle\ v@w'@x) \wedge$
$(\mathit{length}\ p1 > 0 \longrightarrow \mathit{length}\ (v@x) > 0)$
using *assms* **proof** (*induction p1 arbitrary*: *P A z*)
case *Nil*
hence $\forall\, w'\ p'.\ ((P \vdash X \Rightarrow\langle [X]@p2\rangle\ z) \wedge z = []@z@[] \wedge$
$(P \vdash X \Rightarrow\langle [X]@p'\rangle\ w' \longrightarrow P \vdash A \Rightarrow\langle []@[X]@p'\rangle\ ([]@w'@[])) \wedge$
$(\mathit{length}\ [] > 0 \longrightarrow \mathit{length}\ ([]@[]) > 0))$
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 $\mathit{length}\ (p1@[X]@p2) \geq 1$
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 $\mathit{length}\ (v'@x) > 0$
using *True no_empty* **by** *fast*
hence $P \vdash X \Rightarrow\langle [X]@p2\rangle\ w \wedge z = v'@w@x \wedge (\forall\, w'\ p'.$
$P \vdash X \Rightarrow\langle [X]@p'\rangle\ w' \longrightarrow P \vdash A \Rightarrow\langle A\#p1@[X]@p'\rangle\ (v'@w'@x)) \wedge$
$(\mathit{length}\ (A\#p1) > 0 \longrightarrow \mathit{length}\ (v'@x) > 0)$

```
    using vwx 1 BC ‹v' = _› by simp
   thus ?thesis
    using 0 by auto
 next
   case False
   then obtain v w x where vwx: (P ⊢ X ⇒⟨[X]@p2⟩ w) ∧ a = v@w@x ∧
       (∀ w' p'. P ⊢ X ⇒⟨[X]@p'⟩ w' ⟶ P ⊢ B ⇒⟨p1@[X]@p'⟩ (v@w'@x))
    using Cons.IH BC by blast
  hence 1: ∀ w' p'. P ⊢ X ⇒⟨[X]@p'⟩ w' ⟶ P ⊢ A ⇒⟨A#p1@[X]@p'⟩ (v@w'@x@b)
    using BC left[of A B C P] by fastforce
   hence P ⊢ X ⇒⟨[X]@p2⟩ w ∧ z = v@w@x@b ∧ (∀ w' p'.
       P ⊢ X ⇒⟨[X]@p'⟩ w' ⟶ P ⊢ A ⇒⟨A#p1@[X]@p'⟩ (v@w'@x@b)) ∧
       (length (A#p1) > 0 ⟶ 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 ⊢ A ⇒⟪p1@[X]@p2⟫ z
 shows ∃ v w x. P ⊢ X ⇒⟪[X]@p2⟫ w ∧ z = v@w@x ∧
     (∀ w' p'. P ⊢ X ⇒⟨[X]@p'⟩ w' ⟶ P ⊢ A ⇒⟨p1@[X]@p'⟩ v@w'@x)
using assms proof (induction p1 arbitrary: P A z)
 case Nil
 hence ∀ w' p'. P ⊢ X ⇒⟨[X]@p'⟩ w' ⟶ P ⊢ A ⇒⟨[]@[X]@p'⟩ ([]@w'@[])
  using path_first_step lpath_path by fastforce
 moreover have P ⊢ X ⇒⟪[X]@p2⟫ z ∧ 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) ≥ 1
  by simp
 hence ∃ B C p' a b. (A, [Nt B, Nt C]) ∈ P ∧ z = a@b ∧
     ((P ⊢ B ⇒⟪p1@[X]@p2⟫ a ∧ P ⊢ C ⇒⟪p'⟫ b ∧ length (p1@[X]@p2) ≥
length p') ∨
      (P ⊢ B ⇒⟪p'⟫ a ∧ P ⊢ C ⇒⟪p1@[X]@p2⟫ b ∧ length (p1@[X]@p2) ≥
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]) ∈ P ∧ z = a@b ∧
     ((P ⊢ B ⇒⟪p1@[X]@p2⟫ a ∧ P ⊢ C ⇒⟪p'⟫ b ∧ length (p1@[X]@p2) ≥
length p') ∨
      (P ⊢ B ⇒⟪p'⟫ a ∧ P ⊢ C ⇒⟪p1@[X]@p2⟫ b ∧ length (p1@[X]@p2) ≥
```

length p′))
by *blast*
then show *?case*
proof (*cases* $P \vdash B \Rightarrow\langle\!|p1@[X]@p2|\!\rangle\ a \wedge P \vdash C \Rightarrow\langle\!|p'|\!\rangle\ b \wedge$ *length* (*p1*@[*X*]@*p2*) ≥ *length p′*)
case *True*
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* **by** *blast*
hence $P \vdash X \Rightarrow\langle\!|[X]@p2|\!\rangle\ w \wedge z = v@w@x@b \wedge$
$(\forall w'\ p'.\ P \vdash X \Rightarrow\langle[X]@p'\rangle\ w' \longrightarrow P \vdash A \Rightarrow\langle A\#p1@[X]@p'\rangle\ v@w'@x@b)$
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 ($P \vdash B \Rightarrow\langle\!|p'|\!\rangle\ a \wedge P \vdash C \Rightarrow\langle\!|p1@[X]@p2|\!\rangle\ b \wedge$ *length* (*p1*@[*X*]@*p2*) ≥ *length p′*)
using *BC* **by** *blast*
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*
then obtain *v′* **where** $v' = a@v$
by *simp*
hence $P \vdash X \Rightarrow\langle\!|[X]@p2|\!\rangle\ w \wedge z = v'@w@x \wedge$
$(\forall w'\ p'.\ P \vdash X \Rightarrow\langle[X]@p'\rangle\ w' \longrightarrow P \vdash A \Rightarrow\langle A\#p1@[X]@p'\rangle\ v'@w'@x)$
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 ∈ Lang P S
    and length z ≥ 2^(m+1)
  shows ∃ u v w x y . z = u@v@w@x@y ∧ length (v@w@x) ≤ 2^(m+1) ∧ length
(v@x) ≥ 1 ∧ (∀ i. u@(v^^i)@w@(x^^i)@y ∈ Lang P S)
proof −
  obtain p' where p': P ⊢ S ⇒⟨p'⟩ z
    using assms Lang_def[of P S] path_if_derives[of P S] by blast
  then obtain lp where lp: P ⊢ S ⇒⟪lp⟫ z
    using path_lpath[of P] by blast
  hence 1: set lp ⊆ 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 $\forall i.\ P \vdash A \Rightarrow \langle([A]@p2)^{\frown\frown}(Suc\ i)\ @\ [A]@p3\rangle\ v^{\frown\frown}(Suc\ i)\ @\ w\ @\ x^{\frown\frown}(Suc\ i)$
using *vwx inner_aux*[*of P A*] **by** *blast*
hence $\forall i.\ P \vdash S \Rightarrow \langle l@p1@(([A]@p2)^{\frown\frown}(Suc\ i))\ @[A]@p3\rangle\ u@\ (v^{\frown\frown}(Suc\ i))\ @\ w\ @\ (x^{\frown\frown}(Suc\ i))\ @y$
using *uy* **by** *fastforce*
moreover have $P \vdash S \Rightarrow \langle l@p1@(([A]@p2)^{\frown\frown}0)\ @[A]@p3\rangle\ u@\ (v^{\frown\frown}0)\ @\ w\ @\ (x^{\frown\frown}0)\ @y$
using *vwx uy* **by** *auto*
ultimately show $P \vdash S \Rightarrow \langle l@p1@\ (([A]@p2)^{\frown\frown}i)\ @[A]@p3\rangle\ u@(v^{\frown\frown}i)@w@(x^{\frown\frown}i)@y$
by (*induction i*) *simp_all*
qed
hence $\forall i.\ u@(v^{\frown\frown}i)@w@(x^{\frown\frown}i)@y \in Lang\ P\ S$
unfolding *Lang_def* **using** *assms*(*1*) *assms*(*2*) *derives_if_path*[*of P S*] **by** *blast*
hence $z = u@v@w@x@y \wedge length\ (v@w@x) \leq 2\hat{}(m+1) \wedge 1 \leq length\ (v@x) \wedge (\forall\ i.\ u@(v^{\frown\frown}i)@w@(x^{\frown\frown}i)@\ y \in Lang\ P\ S)$
using *vwx uy* ‹*length vwx* $\leq 2\ \hat{}\ (m + 1)$› **by** (*simp add*: *Suc_leI*)
then show *?thesis*
by *blast*
qed

abbreviation *pumping_property L n* $\equiv \forall z \in L.\ length\ z \geq n \longrightarrow$
$(\exists u\ v\ w\ x\ y.\ z = u\ @\ v\ @\ w\ @\ x\ @\ y \wedge length\ (v@w@x) \leq n \wedge length\ (v@x) \geq 1$
$\wedge\ (\forall i.\ u\ @\ v^{\frown\frown}i\ @\ w\ @\ x^{\frown\frown}i\ @\ y \in L))$

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

theorem *Pumping_Lemma*:
assumes *finite* (*P* :: (*'n*::*fresh0*,*'t*)*Prods*)
shows $\exists n.$ *pumping_property* (*Lang P S*) *n*
proof −
obtain *ps'* :: (*'n*,*'t*)*Prods* **where** *ps'*: *finite ps' CNF*(*ps'*) *Lang ps' S* = *Lang P S* − {[]}
using *cnf_exists*[*of P S*, *OF* ‹*finite P*›] **by** *auto*
have *P'*: *CNF ps' finite ps'* **using** *ps'*(*1*,*2*) **by** *auto*
from *Pumping_Lemma_CNF*[*OF P'*, *of S*] **obtain** *n* **where**
pump: *pumping_property* (*Lang ps' S*) *n* **by** *blast*
then have *pumping_property* (*Lang ps' S*) (*Suc n*)
by (*metis Suc_leD nle_le*)
then have *pumping_property* (*Lang P S*) (*Suc n*)
using *ps'*(*3*) **by** (*metis Diff_iff list.size*(*3*) *not_less_eq_eq singletonD zero_le*)
then show *?thesis* **by** *blast*
qed

end

12 $a^n b^n c^n$ 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 $x@y = [a]^{\frown\frown}n\ @\ [b]^{\frown\frown}n\ @\ [c]^{\frown\frown}n$ *length* $x \geq n$
shows *count_list y a = 0*

this prof is easier than $[\![?x\ @\ ?y = [a]^{?n}\ @\ [b]^{?n}\ @\ [c]^{?n};\ ?n \leq length\ ?y]\!] \Longrightarrow count\_list\ ?x\ c = 0$ 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* $(x@y) = [a]^{\frown\frown}n$
using *assms* **by** *simp*
then have *count_take_n*: *count_list* $(take\ n\ (x@y))\ a = n$
by (*simp*)
have $\exists\ z.\ x = take\ n\ (x@y)\ @\ z$
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 $u@w@y = [a]^{\frown\frown}n\ @\ [b]^{\frown\frown}n\ @\ [c]^{\frown\frown}n$ *length* $w \leq n$
shows *count_list w a = 0* $\vee$ *count_list w c = 0*

This lemma uses *count_list w a = 0* $\vee$ *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* $\geq 2*n$ **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]^^n @ [b]^^n @ [c]^^n v@x ≠ []
 shows count_list (v@x) a ≠ 0 ∨ count_list (v@x) b ≠ 0 ∨ count_list (v@x) c
≠ 0
proof −
 have set: set ([a]^^n @ [b]^^n @ [c]^^n) = {a,b,c} using assms pow_list_single_Nil_iff
   by (fastforce simp add: pow_list_single)
 show ?thesis proof (cases  v≠[])
  case True
  then have ∃ d∈set([a]^^n @ [b]^^n @ [c]^^n). count_list v d ≠ 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 ≠ 0 ∨ count_list v b ≠ 0 ∨ count_list v c ≠0
    using set by simp
  then show ?thesis
    by force
 next
  case False
  then have x≠[] using assms
    by fast
  then have ∃ d∈set ([a]^^n @ [b]^^n @ [c]^^n). count_list x d ≠ 0
  proof −
    have ∃ d∈set ([a] ^^ n) ∪ (set ([b] ^^ n) ∪ set ([c] ^^ n)). ya ≠ d ⟶ 0 <
count_list ys d
      if u @ v @ w @ ya # ys @ y = [a] ^^ n @ [b] ^^ n @ [c] ^^ 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 ≠ []› by (auto simp: neq_Nil_conv)
  qed
```

```
    then have count_list x a ≠ 0 ∨ count_list x b ≠ 0 ∨ count_list x c ≠ 0
      using set by simp
    then show ?thesis
      by force
  qed
qed

lemma not_ex_y_count:
  assumes i≠k ∨ k≠j ∨ i≠j count_list w a = i count_list w b = k count_list w
c = j
  shows ¬(EX y. w = [a]^^y @ [b]^^y @ [c]^^y)
 proof
   assume EX y. w = [a]^^y @ [b]^^y @ [c]^^y
   then obtain y where y: w = [a]^^y @ [b]^^y @ [c]^^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 $u @ v^0 @ w @ x^0 @ y$ is not in the language $\bigcup n. \{[a]^n @ [b]^n @ [c]^n\}$ 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*: $((count\_list\ (v@x)\ a) \neq 0) \vee (count\_list\ (v@x)\ b \neq 0) \vee (count\_list\ (v@x)\ c \neq 0)$
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* $(v@x)$ $b \neq 0 \vee count\_list\ (v@x)\ c \neq 0$ **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* $(u@w@y)$ $b \neq n \vee count\_list\ (u@w@y)\ c \neq n$ **using** *vx_b_or_c_not0 count_word_b count_word_c*
by *auto*
then have $(count\_list\ (u@w@y)\ a \neq count\_list\ (u@w@y)\ b) \vee (count\_list\ (u@w@y)\ c \neq count\_list\ (u@w@y)\ a)$ **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*: $(count\_list\ (v@x)\ a \neq 0) \vee (count\_list\ (v@x)\ b \neq 0)$ **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* $(u@w@y)$ $a \neq n \vee count\_list\ (u@w@y)\ b \neq n$ **using** *vx_a_or_b_not0 count_word_a count_word_b*
by *force*
then have $(count\_list\ (u@w@y)\ a \neq count\_list\ (u@w@y)\ b) \vee (count\_list\ (u@w@y)\ c \neq count\_list\ (u@w@y)\ a)$ **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* $(P :: ('n::fresh0, 'a)Prods)$
shows *Lang P S* $\neq (\bigcup n.\ \{[a]^{\frown\frown}n @ [b]^{\frown\frown}n @ [c]^{\frown\frown}n\})$ (**is** $\neg$ *?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

13 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 ⊆ ?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] ∨ (map Tm w::('n,
'a)syms) = ([Tm a]^^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 ⊢ 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] ∨ u @
[Nt X] @ v = ([Tm a]^^n)
          using ‹A=X› step by auto
        have notin: Nt X ∉ set ([Tm a]^^n) by (simp add: pow_list_single)
        then have u @ [Nt X] @ v = ([Tm a]^^n) @ [Nt X]
          using n_src append_Cons in_set_conv_decomp by metis
        then have uv_eq: u = ([Tm a]^^n) ∧ 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] ∨ 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]^^(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]⌢n') @ [Nt X] ∨
((map Tm w)::('n, 'a)syms) = ([Tm a]⌢n') by auto
    have Nt X ∉ set (map Tm w) by auto
     then have ((map Tm w)::('n, 'a)syms) = ([Tm a]⌢n') using n'_src by
fastforce
    have map Tm ([a]⌢n') = ([Tm a]⌢n') by (simp add: map_concat)
      then have w = [a]⌢n' using ‹map Tm w = ([Tm a]⌢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 $\exists n.\ map\ Tm\ w = ([Tm\ a]^{\frown\frown}n) @ [Nt\ X] @ ([Tm\ b]^{\frown\frown}n) \vee (map\ Tm\ w::('n, 'a)syms) = ([Tm\ a]^{\frown\frown}n) @ ([Tm\ b]^{\frown\frown}n)$
proof(*induction rule*: *derives_induct*)
case *base*
have $[Nt\ X] = ([Tm\ a]^{\frown\frown}0) @ [Nt\ X] @ ([Tm\ b]^{\frown\frown}0)$ **by** *auto*
then show *?case* **by** *fast*
next
case (*step u A v w'*)
then have $A{=}X$ **using** *P_def* **by** *auto*
have $P \vdash u @ [Nt\ X] @ v \Rightarrow u @ w' @ v$ **using** ‹$A{=}X$› *derive.intros step* **by** *fast*
obtain *n* **where** *n_src*: $u @ [Nt\ X] @ v = ([Tm\ a]^{\frown\frown}n) @ [Nt\ X] @ ([Tm\ b]^{\frown\frown}n) \vee u @ [Nt\ X] @ v = ([Tm\ a]^{\frown\frown}n) @ ([Tm\ b]^{\frown\frown}n)$
using ‹$A{=}X$› *step* **by** *auto*
have *notin2*: $Nt\ X \notin set\ ([Tm\ a]^{\frown\frown}n) \wedge Nt\ X \notin set\ ([Tm\ b]^{\frown\frown}n)$
by (*simp add*: *pow_list_single*)
then have *notin*: $Nt\ X \notin set\ (([Tm\ a]^{\frown\frown}n) @ ([Tm\ b]^{\frown\frown}n))$ **by** *simp*
then have *uv_split*: $u @ [Nt\ X] @ v = ([Tm\ a]^{\frown\frown}n) @ [Nt\ X] @ ([Tm\ b]^{\frown\frown}n)$
by (*metis n_src append_Cons in_set_conv_decomp*)
have *u_eq*: $u = ([Tm\ a]^{\frown\frown}n)$
by (*metis* (*no_types*, *lifting*) *uv_split notin2 Cons_eq_appendI append_Cons_eq_iff eq_Nil_appendI*)
then have *v_eq*: $v = ([Tm\ b]^{\frown\frown}n)$
by (*metis* (*no_types*, *lifting*) *uv_split notin2 Cons_eq_appendI append_Cons_eq_iff eq_Nil_appendI*)
have $w' = [Tm\ a, Nt\ X, Tm\ b] \vee w' = []$ **using** *step*(*2*) *P_def* **by** *auto*
then show *?case*
proof
assume $w' = [Tm\ a, Nt\ X, Tm\ b]$
then have $u @ w' @ v = ([Tm\ a]^{\frown\frown}n) @ [Tm\ a, Nt\ X, Tm\ b] @ ([Tm\ b]^{\frown\frown}n)$ **using** *u_eq v_eq* **by** *simp*
then have $u @ w' @ v = ([Tm\ a]^{\frown\frown}(Suc\ n)) @ [Nt\ X] @ ([Tm\ b]^{\frown\frown}(Suc\ n))$
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: $(map\ Tm\ w) = ([Tm\ a]^{\frown\frown}n') @ [Nt\ X] @ ([Tm\ b]^{\frown\frown}n') \vee ((map\ Tm\ w)::('n, 'a)syms) = ([Tm\ a]^{\frown\frown}n') @ ([Tm\ b]^{\frown\frown}n')$ **by** *auto*
have $Nt\ X \notin set\ (map\ Tm\ w)$ **by** *auto*
then have *w_ab*: $((map\ Tm\ w)::('n, 'a)syms) = ([Tm\ a]^{\frown\frown}n') @ ([Tm\ b]^{\frown\frown}n')$ **using** n'_src **by** *fastforce*
have *map_a*: $([Tm\ a]^{\frown\frown}n') = map\ Tm\ ([a]^{\frown\frown}n')$ **by** (*simp add*: *map_concat*)
have *map_b*: $([Tm\ b]^{\frown\frown}n') = map\ Tm\ ([b]^{\frown\frown}n')$ **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 $w \in ?L$ **by** *auto*
qed
next
show $?L \subseteq Lang\ P\ X$
proof
fix w
assume $w \in ?L$
then obtain n **where** *n_src*: $w = [a]@@n @ [b]@@n$ **by** *blast*

have *Xa*: $P \vdash [Nt\ X] \Rightarrow(n) ([Tm\ a]@@n) @ [Nt\ X] @ ([Tm\ b]@@n)$
using *P_def deriven_same_repl*[*of X* [*Tm a*] [*Tm b*] _ _ [] []] **by** *simp*
have *Xε*: $P \vdash ([Tm\ a]@@n) @ [Nt\ X] @ ([Tm\ b]@@n) \Rightarrow ([Tm\ a]@@n) @ ([Tm\ b]@@n)$
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 ∧ x1 = y1 ∧ 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≠b b≠c c≠a*
shows ∃ *L1 L2* :: *'a list set*.
CFL TYPE((('n1 + 'n1) option) L1 ∧ CFL TYPE((('n2 + 'n2) option) L2
∧ (∄(P::('x::fresh0,'a)Prods) S. Lang P S = L1 ∩ L2 ∧ finite P)
proof −
let *?anbn* = ⋃ *n*. {[*a*]@@*n* @ [*b*]@@*n*}
let *?cm* = ⋃ *n*. {[*c*]@@*n*}
let *?an* = ⋃ *n*. {[*a*]@@*n*}
let *?bmcm* = ⋃ *n*. {[*b*]@@*n* @ [*c*]@@*n*}
let *?anbncm* = ⋃ *n*. ⋃ *m*. {[*a*]@@*n* @ [*b*]@@*n* @ [*c*]@@*m*}
let *?anbmcm* = ⋃ *n*. ⋃ *m*. {[*a*]@@*n* @ [*b*]@@*m* @ [*c*]@@*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* = (⋃ *n*. {[*a*]@@*n*@[*b*]@@*n*@[*c*]@@*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

14 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* $\notin$ *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* ∉ *Syms* (*set ps*) ⟹ *substP s u ps* = *ps*
by (*induction ps*) (*auto simp add*: *substP_def substs_skip*)

lemma *substP_rev*: *Nt B* ∉ *Syms* (*set ps*) ⟹ *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 $(B',v') \in set\ ps$
using *step* **by** *auto*
then have $(B',\ substsNt\ B\ v\ v') \in set\ (substP\ (Nt\ B)\ v\ ps)$
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 $A \neq B$
and $B \notin lhss\ ps$
and $Nt\ B \notin set\ v$
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

15 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 ^^ n) x) ≤ m x
   then show ?thesis
   proof (cases m ((f ^^ Suc n) x) < m ((f ^^ n) x))
    case True
    then show ?thesis using a1 by auto
   next
    case False
    with assms have (f ^^ Suc n) x = (f ^^ 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 (set 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* (*set 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*) ∈ *set*(*binarize ps*) ⟹ *length w* ≤ *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* ∨ (∃ *A ps'' B u s*. *set ps* = {(*A*, *s*#*u*)} ∪ *set ps''* ∧ *set* (*binarize1 ps' ps*) = {(*A*,[*s*,*Nt B*]),(*B*,*u*)} ∪ *set ps''* ∧ *Nt B* ∉ *Syms* (*set 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*)} ∪ *set ps''* ∧ *set* (*binarize1 ps' ps*) = {(*A*, [*s*, *Nt B*]), (*B*, *u*)} ∪ *set ps''* ∧ *Nt B* ∉ *Syms* (*set ps'*)
using *2* **by** *blast*
from *defs* **have** *wit*: *set* ((*C*, []) # *ps*) = {(*A*, *s* # *u*)} ∪ *set* ((*C*, []) # *ps''*) **by** *simp*
from *defs* **have** *wit2*: *set* (*binarize1 ps'* ((*C*, []) # *ps*)) = {(*A*, [*s*, *Nt B*]), (*B*, *u*)} ∪ *set* ((*C*, []) # *ps''*) **by** *simp*
from *defs* **have** *wit3*: *Nt B* ∉ *Syms* (*set 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* ≤ *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*)} ∪ *set ps''* ∧ *set* (*binarize1 ps' ps*) = {(*A*, [*s*, *Nt B*]), (*B*, *v*)} ∪ *set ps''* ∧ *Nt B* ∉ *Syms* (*set ps'*)
using *3* **by** *blast*
from *defs* **have** *wit*: *set* ((*C*, *s0* # *u*) # *ps*) = {(*A*, *s* # *v*)} ∪ *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*)} ∪ *set* ((*C*, *s0* # *u*) # *ps''*) **by** *simp*
from *defs* **have** *wit3*: *Nt B* ∉ *Syms* (*set 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* (*set 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* ∈ *lhss ps*
shows *set ps* ⊢ [*Nt A*] ⇒∗ *map Tm x* ⟷ *set* (*binarize' ps*) ⊢ [*Nt A*] ⇒∗ *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*)} ∪ *set ps''* ∧ *set* (*binarize1 ps ps*) = {(*C*, [*s*, *Nt B*]), (*B*, *u*)} ∪ *set ps''* ∧ *Nt B* ∉ *Syms* (*set ps*)
by (*meson binarize1_cases*)
from *defs* **have** *a_not_b*: *C* ≠ *B* **unfolding** *Syms_def* **by** *fast*
from *defs assms* **have** *a1*: *A* ≠ *B* **unfolding** *Lhss_def Syms_def* **by** *auto*
from *defs* **have** *a2*: *Nt B* ∉ *set* (*map Tm x*) **by** *auto*
from *defs* **have** *a3*: *Nt B* ∉ *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* ∉ *lhss* ((*C*, [*s*, *Nt B*]) # *ps''*) **unfolding** *Lhss_def Syms_def* **by** *auto*
from *defs* **have** *notB*: *Nt B* ∉ *Syms* (*set 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*)} ∪ *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* ⊢ [*Nt A*] ⇒∗ *map Tm x*) = (*set* (*binarize1 ps ps*) ⊢ [*Nt A*] ⇒∗ *map Tm x*)
  by (*simp add*: *derives_inlining insert_commute*)
qed *simp*

lemma *lhss_binarize1*:
 lhss ps ⊆ *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* (*set 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 (set ps)
    shows  Nt A ∈ Syms (set (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 (set 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 $A \notin lhss\ ps$
and $A \in Nts\ (set\ ps)$
shows $A \notin lhss\ ((binarize'^{\frown\frown} n)\ ps) \wedge A \in Nts\ (set\ ((binarize'^{\frown\frown} n)\ ps))$
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 $A \notin lhss\ ps$
and $A \in Nts\ (set\ ps)$
shows $A \notin lhss\ (binarize\ ps) \wedge A \in Nts\ (set\ (binarize\ ps))$
unfolding *binarize_def* **using** *binarize_lhss_nts'n*[*OF assms*] **by** *simp*

lemma *binarize_nts'n*:
assumes $A \in Nts\ (set\ ps)$
shows $A \in Nts\ (set\ ((binarize'\ ^{\frown\frown}\ n)\ ps))$
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 $A \in Nts\ (set\ ps)$
shows $A \in Nts\ (set\ (binarize\ ps))$
unfolding *binarize_def* **using** *assms binarize_nts'n* **by** *blast*

lemma *lang_binarize*:
assumes $A \in Nts\ (set\ ps)$
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

16 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 = (∀ (*A*,*w*) ∈ *ps*. ∃ *u*. *w* = *map Tm u* ∨ (∃ *B*. *w* = *map Tm u* @ [*Nt B*]))

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

$\vee$ ($\exists\, u.\ w = map\ Tm\ u \vee (\exists\, B.\ w = map\ Tm\ u\ @\ [Nt\ B] \wedge length\ u > 0)$)
  using *rlin_def* **by** *fastforce*
 with *elem* **have** ($\exists\, u.\ w = map\ Tm\ u \vee (\exists\, B.\ w = map\ Tm\ u\ @\ [Nt\ B] \wedge length\ u \leq 0)$)
 $\vee$ ($\exists\, u.\ w = map\ Tm\ u \vee (\exists\, B.\ w = map\ Tm\ u\ @\ [Nt\ B] \wedge length\ u > 0)$) **by** *auto*
 hence ($\exists\, u.\ w = map\ Tm\ u \vee (\exists\, B.\ w = map\ Tm\ u\ @\ [Nt\ B] \wedge length\ u = 0)$)
 $\vee$ ($\exists\, u.\ w = map\ Tm\ u \vee (\exists\, B.\ w = map\ Tm\ u\ @\ [Nt\ B] \wedge length\ u > 0)$) **by** *simp*
 hence ($\exists\, u.\ w = map\ Tm\ u \vee (\exists\, B.\ w = [Nt\ B])$)
 $\vee$ ($\exists\, u.\ w = map\ Tm\ u \vee (\exists\, B.\ w = map\ Tm\ u\ @\ [Nt\ B] \wedge length\ u > 0)$) **by** *auto*
 hence ($\exists\, B.\ w = [Nt\ B]$) $\vee$ ($\exists\, u.\ w = map\ Tm\ u \vee (\exists\, B.\ w = map\ Tm\ u\ @\ [Nt\ B] \wedge length\ u > 0)$) **by** *blast*
 thus *?thesis* **.**
qed

16.1 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* (*set 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* (*set 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* (*set 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* (*set 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* (*set 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* (*set 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* (*set 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* (*set ps'*) **by** *simp*
from *wit wit2 wit3* **show** *?thesis* **by** *blast*
qed *simp*
qed
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* **proof** (*cases finalize1 ps ps = ps*)
case *False*
then obtain *C w ps'' B* **where** *defs*: *set ps* = {(*C*, *w*)} ∪ *set ps''* ∧ *set* (*finalize1 ps ps*) = {(*C*, *w* @ [*Nt B*]), (*B*, [])} ∪ *set ps''* ∧ *Nt B* ∉ *Syms* (*set ps*)
by (*meson finalize1_cases*)
from *defs* **have** *a_not_b*: *C* ≠ *B* **unfolding** *Syms_def* **by** *fast*
from *defs assms* **have** *a1*: *A* ≠ *B* **unfolding** *Lhss_def Syms_def* **by** *auto*
from *defs* **have** *a2*: *Nt B* ∉ *set* (*map Tm x*) **by** *auto*
from *defs* **have** *a3*: *Nt B* ∉ *set* [] **by** *simp*
from *defs* **have** *set ps* = *set* ((*C*, *w*) # *ps''*) **by** *simp*
with *defs a_not_b* **have** *a4*: *B* ∉ *lhss* ((*C*, *w* @ [*Nt B*]) # *ps''*)
unfolding *Lhss_def Syms_def* **by** *auto*
from *defs* **have** *notB*: *Nt B* ∉ *Syms* (*set 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* ∉ *Syms* (*set ps*) **unfolding** *Syms_def* **by** *meson*
from *defs* **have** *s2*: (*C*,*w*) ∈ *set ps* **by** *blast*
from *s1 s2* **have** *b_notin_w*: *Nt B* ∉ *set w* **unfolding** *Syms_def* **by** *fastforce*
from *defs* **have** *set ps* = {(*C*, *w*)} ∪ *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* ⊢ [*Nt A*] ⇒∗ *map Tm x* ⟷ *set* (*finalize1 ps ps*) ⊢ [*Nt A*] ⇒∗ *map Tm x*
  by (*simp add*: *derives_inlining insert_commute*)
qed *simp*

lemma *lhss_finalize1*:
 lhss ps ⊆ *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 (set ps)
    shows  Nt A ∈ Syms (set (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 (set ps)
  shows   A ∈ Nts (set ((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 (set ps)
  shows   A ∈ Nts (set (finalize ps))
  unfolding finalize_def using finalize_nts'n[OF assms] by simp

lemma finalize_lhss_nts1:
  assumes A ∉ lhss ps
      and A ∈ Nts (set 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 $A \notin lhss\ ps$
and $A \in Nts\ (set\ ps)$
shows $A \notin lhss\ ((finalize'^{\frown\frown}n)\ ps) \wedge A \in Nts\ (set\ ((finalize'^{\frown\frown}n)\ ps))$
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 $A \notin lhss\ ps$
and $A \in Nts\ (set\ ps)$
shows $A \notin lhss\ (finalize\ ps) \wedge A \in Nts\ (set\ (finalize\ ps))$
unfolding *finalize_def* **using** *finalize_lhss_nts'n*[*OF assms*] **by** *simp*

lemma *lang_finalize*:
assumes $A \in Nts\ (set\ ps)$
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: ∃ v B. s0 # u = map Tm v @ [Nt B]
    by (simp add: rlin_noterm_def)
  with a1 have 2: ∃ B. s0 # u = [Nt B] ∨ (∃ 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 ∃ B. s0 = Nt B ∧ u = [] ∨ (∃ a. s0 = Tm a) ∧ 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 (set 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 ps'
    and uppr_ps': Unit_elim_rel ps' ps
  shows rlin2 ps
proof −
  from rlinbin have rlin2 (ps' − {(A,w) ∈ ps'. ∃ B. w = [Nt B]})
```

using *rlin2_def rlin_bin_def* **by** *fastforce*
hence *rlin2* (*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 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* (*set* (*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* ((*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* (*Unit_elim* (*set* (*binarize* (*finalize ps*))))
using *Unit_elim_correct uppr_rlin2* **by** *blast*
thus *rlin2* ((*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*: *Lang_Unit_elim finalize_nts lang_binarize lang_finalize rlin2_of_rlin_def*)

16.2 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 $P \vdash [Nt\ A] \Rightarrow* map\ Tm\ w\ @\ [Tm\ x,\ Nt\ B]$
shows $\exists\, C.\ P \vdash [Nt\ A] \Rightarrow* map\ Tm\ w\ @\ [Nt\ C] \wedge (C,[Tm\ x,\ Nt\ B]) \in P$
proof −
from *assms(2)* **have** $\exists\, v.\ P \vdash [Nt\ A] \Rightarrow* v \wedge P \vdash v \Rightarrow map\ Tm\ w\ @\ [Tm\ x,\ Nt\ B]$
using *rtranclp.cases* **by** *fastforce*
then obtain *v* **where** *v_star*: $P \vdash [Nt\ A] \Rightarrow* v$ **and** *v_step*: $P \vdash v \Rightarrow map\ Tm\ w\ @\ [Tm\ x,\ Nt\ B]$ **by** *blast*
from *v_step* **have** $\exists\, u1\ u2\ C\ \alpha.\ v = u1\ @\ [Nt\ C]\ @\ u2 \wedge map\ Tm\ w\ @\ [Tm\ x,\ Nt\ B] = u1\ @\ \alpha\ @\ u2 \wedge (C,\alpha) \in P$
using *derive.cases* **by** *fastforce*
then obtain *u1 u2 C* α **where** *v_def*: $v = u1\ @\ [Nt\ C]\ @\ u2$ **and** *w_def*: $map\ Tm\ w\ @\ [Tm\ x,\ Nt\ B] = u1\ @\ \alpha\ @\ u2$
and *C_prod*: $(C,\alpha) \in P$ **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** $map\ Tm\ w\ @\ [Tm\ x,\ Nt\ B] = map\ Tm\ wa\ @\ \alpha$ **by** *simp*
then have $map\ Tm\ (w\ @\ [x])\ @\ [Nt\ B] = map\ Tm\ wa\ @\ \alpha$ **by** *simp*
then have $\alpha \neq []$
by (*metis append.assoc append.right_neutral list.distinct(1) map_Tm_single_nt*)
with *assms(1) C_prod* **obtain** *d D* **where** $\alpha = [Tm\ d,\ Nt\ D]$
using *rlin2_def* **by** *fastforce*
from *w_def u2_eps* **have** *x_d*: $x = d$
using ‹$\alpha = [Tm\ d,\ Nt\ D]$› **by** *simp*
from *w_def u2_eps* **have** *B_D*: $B = D$
using ‹$\alpha = [Tm\ d,\ Nt\ D]$› **by** *simp*
from *x_d B_D* **have** *alpha_def*: $\alpha = [Tm\ x,\ Nt\ B]$
using ‹$\alpha = [Tm\ d,\ Nt\ D]$› **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*: $P \vdash [Nt\ A] \Rightarrow* map\ Tm\ w\ @\ [Nt\ C]$ **by** *simp*
from *C_prod alpha_def* **have** *2*: $(C,[Tm\ x,\ Nt\ B]) \in P$ **by** *simp*
from *1 2* **show** *?thesis* **by** *auto*
qed

lemma *rlin2_nts_derive_eq*:
assumes *rlin2 P*
and $P \vdash [Nt\ A] \Rightarrow* [Nt\ B]$
shows $A = B$
proof −
from *assms(2)* **have** *star_cases*: $[Nt\ A] = [Nt\ B] \vee (\exists\, w.\ P \vdash [Nt\ A] \Rightarrow w \wedge P \vdash w \Rightarrow* [Nt\ B])$

```
    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 ⊢ [Nt A] ⇒* map Tm w
    shows ∃ B. P ⊢ [Nt A] ⇒* map Tm w @ [Nt B] ∧ (B,[]) ∈ P
proof −
  from assms(2) have ∃ v. P ⊢ [Nt A] ⇒* v ∧ P ⊢ v ⇒ map Tm w
    using rtranclp.cases by force
  then obtain v where v_star: P ⊢ [Nt A] ⇒* v and v_step: P ⊢ v ⇒ map Tm
w by blast
  from v_step have ∃ u1 u2 C α. v = u1 @ [Nt C] @ u2 ∧ map Tm w = 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 = u1 @ α @ u2 and C_prod: (C,α) ∈ P by blast
  have ∄ A. Nt A ∈ set (map Tm w) by auto
  with w_def have ∄ A. Nt A ∈ set α
    by (metis Un_iff set_append)
  then have ∄ a A. α = [Tm a, Nt A] by auto
  with assms(1) C_prod have alpha_eps: α = []
    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 ⊢ [Nt A] ⇒* map Tm w @ [Nt C]
by simp
  from alpha_eps C_prod have 2: (C,[]) ∈ P  by simp
  from 1 2 show ?thesis by auto
```

qed

end

17 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* ⇒ 'n ⇒ 't ⇒ 'n *set* **where**
nxt_rlin2 P A a = {*B*. (*A*, [*Tm a*, *Nt B*]) ∈ *P*}

definition *nxt_rlin2_set* :: ('n,'t)*Prods* ⇒ 'n *set* ⇒ 't ⇒ 'n *set* **where**
nxt_rlin2_set P M a = (⋃ *A*∈*M*. *nxt_rlin2 P A a*)

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

lemma *nxt_rlin2_nts*:
assumes *B*∈*nxt_rlin2 P A a*
shows *B*∈*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* ∈ *nxt_rlin2_set P M a*
shows ∃ *C*∈*M*. *B* ∈ *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* ∈ *nxts_rlin2_set P M w*
shows ∃ *C*∈*M*. *B* ∈ *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*:
$B \in$ *nxts_rlin2_set P M w* $\Longrightarrow$ ($\exists\, A{\in}M.\ P \vdash [Nt\ A] \Rightarrow* map\ Tm\ w\ @\ [Nt\ B]$)
unfolding *nxts_rlin2_set_def* **proof** (*induction w arbitrary*: *B rule*: *rev_induct*)
case *Nil*
hence *1*: $B \in M$ **by** *simp*
have *2*: $P \vdash [Nt\ B] \Rightarrow* map\ Tm\ []\ @\ [Nt\ B]$ **by** *simp*
from *1 2* **show** *?case* **by** *blast*
next
case (*snoc x xs*)
from *snoc.prems* **have** $\exists\, C.\ C \in$ *foldl* (*nxt_rlin2_set P*) *M xs* $\wedge$ (*C*, [*Tm x*, *Nt B*]) $\in P$
unfolding *nxt_rlin2_set_def nxt_rlin2_def* **by** *auto*
then obtain *C* **where** *C_xs*: $C \in$ *foldl* (*nxt_rlin2_set P*) *M xs* **and** *C_prod*: (*C*, [*Tm x*, *Nt B*]) $\in P$ **by** *blast*
from *C_xs* **obtain** *A* **where** *A_der*: $P \vdash [Nt\ A] \Rightarrow* map\ Tm\ xs\ @\ [Nt\ C]$ **and** *A_in*: $A \in M$
using *snoc.IH*[*of C*] **by** *auto*
from *C_prod* **have** $P \vdash [Nt\ C] \Rightarrow [Tm\ x, Nt\ B]$
using *derive_singleton*[*of P Nt C* [*Tm x*, *Nt B*]] **by** *blast*
hence $P \vdash map\ Tm\ xs\ @\ [Nt\ C] \Rightarrow map\ Tm\ xs\ @\ [Tm\ x, Nt\ B]$
using *derive_prepend*[*of P* [*Nt C*] [*Tm x*, *Nt B*] *map Tm xs*] **by** *simp*
hence *C_der*: $P \vdash map\ Tm\ xs\ @\ [Nt\ C] \Rightarrow map\ Tm\ (xs\ @\ [x])\ @\ [Nt\ B]$ **by** *simp*
from *A_der C_der* **have** $P \vdash [Nt\ A] \Rightarrow* map\ Tm\ (xs\ @\ [x])\ @\ [Nt\ B]$ **by** *simp*
with *A_in* **show** *?case* **by** *blast*
qed

lemma *mult_derive_to_nxts*:
assumes *rlin2 P*
shows $A{\in}M \Longrightarrow P \vdash [Nt\ A] \Rightarrow* map\ Tm\ w\ @\ [Nt\ B] \Longrightarrow B \in$ *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 = (∃ Z ∈ nxts_rlin2_set P {A} w. (Z,[]) ∈ P)

theorem accepted_if_Lang:
  assumes rlin2 P
      and w ∈ Lang P A
    shows accepted P A w
proof −
  from assms obtain B where A_der: P ⊢ [Nt A] ⇒* map Tm w @ [Nt B] and
B_in: (B,[]) ∈ P
    unfolding Lang_def using rlin2_tms_eps[of P A w] by auto
  from A_der have B ∈ 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 ∈ Lang P A
proof −
  from assms obtain Z where Z_nxts: Z ∈ nxts_rlin2_set P {A} w and Z_eps:
(Z,[]) ∈ P
    unfolding accepted_def by blast
  from Z_nxts obtain B where B_der: P ⊢ [Nt B] ⇒* map Tm w @ [Nt Z] and
B_in: B ∈ {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

18 Relating Strongly Right-Linear Grammars and Automata

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

18.1 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* ∈ *Nts P*. (*A*,[]) ∈ *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

18.2 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 ⟹ ∃ *P S*::*hf*. *rlin2 P* ∧ *L* = *Lang P S*
by (*metis dfa.dfa_language_eq_Lang dfa.rlin2_prods_dfa regular_def*)

end

19 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* ≥ *1*
and ∀ *i* < *length w*. *w* ! *i* ∈ *P*
and *length w* ≥ *Suc m*
shows ∃ *xs ys zs y*. *w* = *xs* @ [*y*] @ *ys* @ [*y*] @ *zs* ∧ *length* (*xs* @ [*y*] @ *ys* @ [*y*]) ≤ *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* ≤ *length w*
using *Suc_le_length_iff*[*of m aw*] **by** *blast*
show *?case* **proof** (*cases a* ∈ *set w*)
case *True*

```
    hence ¬ 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]) ≤ 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 ≥ Suc m by simp
      from aw_split have a5: length ?xsyys < length aw by simp
      with 1.prems(3) have ∀ i<length ?xsyys. aw ! i ∈ P by simp
      with aw_split have a3: ∀ i < length ?xsyys. ?xsyys ! i ∈ P
        by (metis append_assoc nth_append)
      from 1.prems(2) 1.prems(1) a3 a4 a5 have ∃ xs' ys' zs' y'. ?xsyys = xs' @
[y'] @ ys' @ [y'] @ zs' ∧ length (xs' @ [y'] @ ys' @ [y']) ≤ 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 travers-

ing 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* :: $('n,'t)Prods \Rightarrow 'n \Rightarrow 't\ list \Rightarrow 'n\ list\ set$ **where**
  nxts_nts P A [] = {[]}
| *nxts_nts P A* (*a*#*w*) = ($\bigcup B{\in}nxt\_rlin2\ P\ A\ a.\ (Cons\ B)$ ‘*nxts_nts P B w*)

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

19.1 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 ∀ *e* ∈ *nxts_nts0 P A w*. ∃ *e'* ∈ *nxts_nts P A w*. *e* ! (*Suc i*) = *e'* ! *i*
  unfolding *nxts_nts0_def* **by** *auto*

lemma *nxts_nts_pick_nt*:
  assumes *e* ∈ *nxts_nts P A* (*a*#*w*)
  shows ∃ *C*∈*nxt_rlin2 P A a*. ∃ *e'* ∈ *nxts_nts P C w*. *e* = *C*#*e'*
  using *assms* **by** *auto*

lemma *nxts_nts0_len*:
  ∀ *e* ∈ *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 ∀ *e* ∈ *nxts_nts0 P A w*. *e*!(*Suc i*) ∈ *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* ≤ *length w*
    and *i2* ≤ *length w*
    and *i1* ≤ *i2*
  shows ∀ *e* ∈ *nxts_nts0 P A w*. *e*!*i2* ∈ *nxts_rlin2_set P* {*e*!*i1*} (*drop i1* (*take*

```
i2 w))
proof
  fix e
  assume e ∈ nxts_nts0 P A w
  with assms show e ! i2 ∈ 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' ≤ length w by simp
      from Suc.prems(3) Suc.hyps(2) have i1_i2': i1 ≤ ?i2' by simp
      have IH: e ! ?i2' ∈ 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
```

19.2 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 ∈ Nts P
      and m = card (Nts P)
      and B ∈ nxts_rlin2_set P {A} w
      and length w ≥ m
    shows ∃ x y z C. w = x@y@z ∧ length y ≥ 1 ∧ length (x@y) ≤ m ∧
            C ∈ nxts_rlin2_set P {A} x ∧ C ∈ nxts_rlin2_set P {C} y ∧ B ∈
nxts_rlin2_set P {C} z
proof −
  let ?nts = nxts_nts0 P A w
  obtain e where e_def: e ∈ ?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: ∀ i < length e. e!i ∈ 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 ≥ 1
    using less_eq_Suc_le by fastforce
  from assms(5) e_len have ∃ xs ys zs y. e = xs @ [y] @ ys @ [y] @ zs ∧ length
(xs @ [y] @ ys @ [y]) ≤ Suc m
    using not_distinct[OF assms(3) m_geq_1 e_elem] by simp
```

then obtain *xs ys zs C* **where** *e_split*: $e = xs \mathbin{@} [C] \mathbin{@} ys \mathbin{@} [C] \mathbin{@} zs$ **and** *xy_len*: $length\ (xs \mathbin{@} [C] \mathbin{@} ys \mathbin{@} [C]) \leq Suc\ m$
by *blast*
let *?e1* $= xs \mathbin{@} [C]$ **let** *?e2* $= ys \mathbin{@} [C]$ **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** $**$: $length\ ?y \geq 1$ **by** *simp*
from *xy_len* **have** $***$: $length\ (?x @ ?y) \leq m$ **by** *simp*
have *x_fac*: $?x = take\ (length\ xs)\ w$ **by** *simp*
from $**$ **have** *x_fac2*: $length\ xs \leq length\ w$ **by** *simp*
from *e_split* **have** *x_fac3*: $e\ !\ length\ xs = C$ **by** *simp*
from *e_def x_fac x_fac3* **have** $****$: $C \in nxts\_rlin2\_set\ P\ \{A\}\ ?x$
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*: $length\ xs + length\ ys + 1 \leq length\ w$ **by** *simp*
have *y_fac3*: $length\ xs \leq length\ xs + length\ ys + 1$ **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** $*****$: $C \in nxts\_rlin2\_set\ P\ \{C\}\ ?y$
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** $******$: $B \in nxts\_rlin2\_set\ P\ \{C\}\ ?z$
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 $B \in nxts\_rlin2\_set\ P\ \{A\}\ x$
and $B \in nxts\_rlin2\_set\ P\ \{B\}\ y$
shows $B \in nxts\_rlin2\_set\ P\ \{A\}\ (x @ (y \frown\frown i))$
using *assms* **proof** (*induction i*)
case *0*
thus *?case* **by** (*simp add*: *assms*(*1*))
next
case (*Suc i*)
have $B \in nxts\_rlin2\_set\ P\ \{A\}\ (x @ (y \frown\frown i))$
using *Suc.IH*[*OF assms*] **.**
with *assms*(*2*) **have** $B \in nxts\_rlin2\_set\ P\ \{A\}\ (x @ (y \frown\frown i) @ y)$
using *nxts_trans2*[*of B P A* $x @ (y \frown\frown i)$ *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 $A \in Nts\ P$
and $m = card\ (Nts\ P)$
and *accepted P A w*
and *length* $w \geq m$
shows $\exists\, x\ y\ z.\ w = x@y@z \wedge length\ y \geq 1 \wedge length\ (x@y) \leq m \wedge (\forall\, i.\ accepted\ P\ A\ (x@(y^{\frown\frown}i)@z))$
proof −
from *assms*(*4*) **obtain** *Z* **where** *Z_in*:$Z \in nxts\_rlin2\_set\ P\ \{A\}\ w$ **and** *Z_eps*:$(Z,[]) \in P$
by (*auto simp*: *accepted_def*)
obtain *x y z C* **where** ∗: $w = x@y@z$ **and** ∗∗: *length* $y \geq 1$ **and** ∗∗∗: *length* $(x@y) \leq m$ **and**
1: $C \in nxts\_rlin2\_set\ P\ \{A\}\ x$ **and** *2*: $C \in nxts\_rlin2\_set\ P\ \{C\}\ y$ **and** *3*: $Z \in nxts\_rlin2\_set\ P\ \{C\}\ z$
using *nxts_split_cycle*[*OF assms*(*1*) *assms*(*2*) *assms*(*3*) *Z_in assms*(*5*)] **by** *auto*
have $\forall\, i.\ C \in nxts\_rlin2\_set\ P\ \{A\}\ (x@(y^{\frown\frown}i))$
using *pump_cycle*[*OF 1 2*] **by** *simp*
with *3* **have** $\forall\, i.\ Z \in nxts\_rlin2\_set\ P\ \{A\}\ (x@(y^{\frown\frown}i)@z)$
using *nxts_trans2*[*of C P A*] **by** *fastforce*
with *Z_eps* **have** ∗∗∗∗: $(\forall\, i.\ accepted\ P\ A\ (x@(y^{\frown\frown}i)@z))$
by (*auto simp*: *accepted_def*)
from ∗ ∗∗ ∗∗∗ ∗∗∗∗ **show** *?thesis* **by** *auto*
qed

theorem *pumping_lemma_re_nts*:
assumes *rlin2 P*
and *finite P*
and $A \in Nts\ P$
shows $\exists\, n.\ \forall\, w \in Lang\ P\ A.\ length\ w \geq n \longrightarrow$
$(\exists\, x\ y\ z.\ w = x@y@z \wedge length\ y \geq 1 \wedge length\ (x@y) \leq n \wedge (\forall\, i.\ x@(y^{\frown\frown}i)@z \in Lang\ P\ A))$
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 $\exists\, n.\ \forall\, w \in Lang\ P\ A.\ length\ w \geq n \longrightarrow$
$(\exists\, x\ y\ z.\ w = x@y@z \wedge length\ y \geq 1 \wedge length\ (x@y) \leq n \wedge (\forall\, i.\ x@(y^{\frown\frown}i)@z \in Lang\ P\ A))$

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

20 $a^n b^n$ 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.