Context-Free Grammars and Languages

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

May 29, 2025

Abstract

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

Contents

1 Context-Free Grammars

theory *Context_Free_Grammar*
imports *HOL−Library.Infinite_Typeclass*
begin

definition *fresh* :: (*'n*::*infinite*) *set* ⇒ *'n* **where**
fresh A = (*SOME x*. *x* ∉ *A*)

lemma *fresh_finite*: *finite A* ⟹ *fresh A* ∉ *A*
⟨*proof*⟩

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

lemma *bex_pair_conv*: (∃ (*x*,*y*) ∈ *R*. *P x y*) ⟷ (∃ *x y*. (*x*,*y*) ∈ *R* ∧ *P x y*)
⟨*proof*⟩

lemma *in_image_map_prod*: *fgp* ∈ *map_prod f g* ‘ *R* ⟷ (∃ (*x*,*y*)∈*R*. *fgp* = (*f x*,*g y*))
⟨*proof*⟩

1.1 Symbols and Context-Free Grammars

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

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

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

type_synonym (*'n*,*'t*) *prod* = *'n* × (*'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* ⇒ *bool* **where**
isTm S = (∃ *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 *isTm_simps*[*simp*]:
  ‹*isTm* (*Nt A*) = *False*›
  ‹*isTm* (*Tm a*)›
⟨*proof*⟩

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

lemma *destTm_o_Tm*[*simp*]: ‹*destTm* ∘ *Tm* = *id*›
⟨*proof*⟩

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*)

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

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

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

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

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

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

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

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

lemma *inj_Nt*: *inj Nt*

⟨*proof*⟩

lemma *map_Tm_inject_iff*[*simp*]: *map Tm xs* = *map Tm ys* ⟷ *xs* = *ys*
⟨*proof*⟩

lemma *map_Nt_eq_map_Nt_iff*[*simp*]: *map Nt u* = *map Nt v* ⟷ *u* = *v*
⟨*proof*⟩

lemma *map_Nt_eq_map_Tm_iff*[*simp*]: *map Nt u* = *map Tm v* ⟷ *u* = [] ∧ *v* = []
⟨*proof*⟩

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

lemma *nts_syms_Nil*[*simp*]: *nts_syms* [] = {}
⟨*proof*⟩

lemma *nts_syms_Cons*[*simp*]: *nts_syms* (*a*#*v*) = (*case a of Nt A* ⇒ {*A*} | _ ⇒ {}) ∪ *nts_syms v*
⟨*proof*⟩

lemma *nts_syms_append*[*simp*]: *nts_syms* (*u* @ *v*) = *nts_syms u* ∪ *nts_syms v*
⟨*proof*⟩

lemma *nts_syms_map_Nt*[*simp*]: *nts_syms* (*map Nt w*) = *set w*
⟨*proof*⟩

lemma *nts_syms_map_Tm*[*simp*]: *nts_syms* (*map Tm w*) = {}
⟨*proof*⟩

lemma *in_Nts_iff_in_Syms*: *B* ∈ *Nts P* ⟷ *Nt B* ∈ *Syms P*
⟨*proof*⟩

lemma *Nts_Un*: *Nts* (*P1* ∪ *P2*) = *Nts P1* ∪ *Nts P2*
⟨*proof*⟩

lemma *Nts_Lhss_Rhs_Nts*: *Nts P* = *Lhss P* ∪ *Rhs_Nts P*
⟨*proof*⟩

lemma *Nts_nts_syms*: *w* ∈ *Rhss P A* ⟹ *nts_syms w* ⊆ *Nts P*
⟨*proof*⟩

lemma *Syms_simps*[*simp*]:
Syms {} = {}
Syms(*insert* (*A*,*w*) *P*) = {*Nt A*} ∪ *set w* ∪ *Syms P*
Syms(*P* ∪ *P*′) = *Syms P* ∪ *Syms P*′
⟨*proof*⟩

lemma *Lhss_simps*[*simp*]:

Lhss {} = {}
Lhss(*insert* (*A*,*w*) *P*) = {*A*} ∪ *Lhss P*
Lhss(*P* ∪ *P'*) = *Lhss P* ∪ *Lhss P'*
⟨*proof*⟩

1.1.1 Finiteness Lemmas

lemma *finite_nts_syms*: *finite* (*nts_syms w*)
⟨*proof*⟩

lemma *finite_nts*: *finite*(*nts ps*)
⟨*proof*⟩

lemma *fresh_nts*: *fresh*(*nts ps*) ∉ *nts ps*
⟨*proof*⟩

lemma *finite_nts_prods_start*: *finite*(*nts*(*prods g*) ∪ {*start g*})
⟨*proof*⟩

lemma *fresh_nts_prods_start*: *fresh*(*nts*(*prods g*) ∪ {*start g*}) ∉ *nts*(*prods g*) ∪ {*start g*}
⟨*proof*⟩

lemma *finite_Nts*: *finite P* ⟹ *finite* (*Nts P*)
⟨*proof*⟩

lemma *finite_Rhss*: *finite P* ⟹ *finite* (*Rhss P A*)
⟨*proof*⟩

1.2 Derivations

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

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

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

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

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

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

lemma *DersI*:

assumes $P \vdash [Nt\ A] \Rightarrow* w$ **shows** $w \in Ders\ P\ A$
⟨*proof*⟩

lemma *DersD*:
assumes $w \in Ders\ P\ A$ **shows** $P \vdash [Nt\ A] \Rightarrow* w$
⟨*proof*⟩

lemmas $DersE = DersD[elim\_format]$

definition $Lang :: ('n,'t)Prods \Rightarrow 'n \Rightarrow 't\ list\ set$ **where**
$Lang\ P\ A = \{w.\ P \vdash [Nt\ A] \Rightarrow* map\ Tm\ w\}$

abbreviation $lang :: ('n,'t)prods \Rightarrow 'n \Rightarrow 't\ list\ set$ **where**
$lang\ ps\ A \equiv Lang\ (set\ ps)\ A$

abbreviation $LangS :: ('n,'t)\ Cfg \Rightarrow 't\ list\ set$ **where**
$LangS\ G \equiv Lang\ (Prods\ G)\ (Start\ G)$

abbreviation $langS :: ('n,'t)\ cfg \Rightarrow 't\ list\ set$ **where**
$langS\ g \equiv lang\ (prods\ g)\ (start\ g)$

lemma *Lang_Ders*: $map\ Tm\ `\ (Lang\ P\ A) \subseteq Ders\ P\ A$
⟨*proof*⟩

lemma *Lang_eqI_derives*:
assumes $\bigwedge v.\ R \vdash [Nt\ A] \Rightarrow* map\ Tm\ v \longleftrightarrow S \vdash [Nt\ A] \Rightarrow* map\ Tm\ v$
shows $Lang\ R\ A = Lang\ S\ A$
⟨*proof*⟩

lemma *derive_iff*: $R \vdash u \Rightarrow v \longleftrightarrow (\exists\ (A,w) \in R.\ \exists\, u1\ u2.\ u = u1\ @\ Nt\ A\ \#\ u2 \wedge v = u1\ @\ w\ @\ u2)$
⟨*proof*⟩

lemma *not_derive_from_Tms*: $\neg\ P \vdash map\ Tm\ as \Rightarrow w$
⟨*proof*⟩

lemma *deriven_from_TmsD*: $P \vdash map\ Tm\ as \Rightarrow(n)\ w \Longrightarrow w = map\ Tm\ as$
⟨*proof*⟩

lemma *derives_from_Tms_iff*: $P \vdash map\ Tm\ as \Rightarrow* w \longleftrightarrow w = map\ Tm\ as$
⟨*proof*⟩

lemma *Un_derive*: $R \cup S \vdash y \Rightarrow z \longleftrightarrow R \vdash y \Rightarrow z \vee S \vdash y \Rightarrow z$
⟨*proof*⟩

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*⟩

lemma *derives_Cons_rule*:
assumes *1*: (*A*,*w*) ∈ *R* **and** *2*: *R* ⊢ *w* @ *u* ⇒∗ *v* **shows** *R* ⊢ *Nt A* # *u* ⇒∗ *v*
⟨*proof*⟩

lemma *deriven_mono*: *P* ⊆ *P′* ⟹ *P* ⊢ *u* ⇒(*n*) *v* ⟹ *P′* ⊢ *u* ⇒(*n*) *v*
⟨*proof*⟩

lemma *derives_mono*: *P* ⊆ *P′* ⟹ *P* ⊢ *u* ⇒∗ *v* ⟹ *P′* ⊢ *u* ⇒∗ *v*
⟨*proof*⟩

lemma *Lang_mono*: *P* ⊆ *P′* ⟹ *Lang P A* ⊆ *Lang P′ A*
⟨*proof*⟩

1.2.2 Customized Induction Principles

lemma *deriven_induct*[*consumes 1*, *case_names 0 Suc*]:
assumes *P* ⊢ *xs* ⇒(*n*) *ys*
and *Q 0 xs*
and ⋀*n u A v w*. ⟦ *P* ⊢ *xs* ⇒(*n*) *u* @ [*Nt A*] @ *v*; *Q n* (*u* @ [*Nt A*] @ *v*); (*A*,*w*) ∈ *P* ⟧ ⟹ *Q* (*Suc n*) (*u* @ *w* @ *v*)
shows *Q n ys*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

1.2.3 (De)composing derivations

lemma *derive_append*:
G ⊢ *u* ⇒ *v* ⟹ *G* ⊢ *u*@*w* ⇒ *v*@*w*
⟨*proof*⟩

lemma *derive_prepend*:
G ⊢ *u* ⇒ *v* ⟹ *G* ⊢ *w*@*u* ⇒ *w*@*v*
⟨*proof*⟩

lemma *deriven_append*:
$P \vdash u \Rightarrow(n)\ v \Longrightarrow P \vdash u\ @\ w \Rightarrow(n)\ v\ @\ w$
⟨*proof*⟩

lemma *deriven_prepend*:
$P \vdash u \Rightarrow(n)\ v \Longrightarrow P \vdash w\ @\ u \Rightarrow(n)\ w\ @\ v$
⟨*proof*⟩

lemma *derives_append*:
$P \vdash u \Rightarrow* v \Longrightarrow P \vdash u@w \Rightarrow* v@w$
⟨*proof*⟩

lemma *derives_prepend*:
$P \vdash u \Rightarrow* v \Longrightarrow P \vdash w@u \Rightarrow* w@v$
⟨*proof*⟩

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*⟩

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*⟩

lemma *derives_append_decomp*:
$P \vdash u\ @\ v \Rightarrow* w \longleftrightarrow (\exists\, u'\ v'.\ P \vdash u \Rightarrow* u' \wedge P \vdash v \Rightarrow* v' \wedge w = u'\ @\ v')$
⟨*proof*⟩

lemma *derives_concat*:
$\forall\, i \in set\ is.\ P \vdash f\ i \Rightarrow*\ g\ i \Longrightarrow P \vdash concat(map\ f\ is) \Rightarrow*\ concat(map\ g\ is)$
⟨*proof*⟩

lemma *derives_concat1*:
$\forall\, i \in set\ is.\ P \vdash [f\ i] \Rightarrow*\ g\ i \Longrightarrow P \vdash map\ f\ is \Rightarrow*\ concat(map\ g\ is)$
⟨*proof*⟩

lemma *derive_Cons*:
$P \vdash u \Rightarrow v \Longrightarrow P \vdash a\#u \Rightarrow a\#v$
⟨*proof*⟩

lemma *derives_Cons*:
$R \vdash u \Rightarrow* v \Longrightarrow R \vdash a\#u \Rightarrow* a\#v$
⟨*proof*⟩

lemma *derive_from_empty*[*simp*]:
$P \vdash [] \Rightarrow w \longleftrightarrow False$
⟨*proof*⟩

lemma *deriven_from_empty*[*simp*]:
$P \vdash [] \Rightarrow(n)\ w \longleftrightarrow n = 0 \wedge w = []$
⟨*proof*⟩

lemma *derives_from_empty*[*simp*]:
$\mathcal{G} \vdash [] \Rightarrow* w \longleftrightarrow w = []$
⟨*proof*⟩

lemma *deriven_start1*:
assumes $P \vdash [Nt\ A] \Rightarrow(n)\ map\ Tm\ w$
shows $\exists \alpha\ m.\ n = Suc\ m \wedge P \vdash \alpha \Rightarrow(m)\ (map\ Tm\ w) \wedge (A,\alpha) \in P$
⟨*proof*⟩

lemma *derives_start1*: $P \vdash [Nt\ A] \Rightarrow* map\ Tm\ w \implies \exists \alpha.\ P \vdash \alpha \Rightarrow* map\ Tm\ w \wedge (A,\alpha) \in P$
⟨*proof*⟩

lemma *Lang_empty_if_notin_Lhss*: $A \notin Lhss\ P \Longrightarrow Lang\ P\ A = \{\}$
⟨*proof*⟩

lemma *derive_Tm_Cons*:
$P \vdash Tm\ a\ \#\ u \Rightarrow v \longleftrightarrow (\exists w.\ v = Tm\ a\ \#\ w \wedge P \vdash u \Rightarrow w)$
⟨*proof*⟩

lemma *deriven_Tm_Cons*:
$P \vdash Tm\ a\ \#\ u \Rightarrow(n)\ v \longleftrightarrow (\exists w.\ v = Tm\ a\ \#\ w \wedge P \vdash u \Rightarrow(n)\ w)$
⟨*proof*⟩

lemma *derives_Tm_Cons*:
$P \vdash Tm\ a\ \#\ u \Rightarrow* v \longleftrightarrow (\exists w.\ v = Tm\ a\ \#\ w \wedge P \vdash u \Rightarrow* w)$
⟨*proof*⟩

lemma *derives_Tm*[*simp*]: $P \vdash [Tm\ a] \Rightarrow* w \longleftrightarrow w = [Tm\ a]$
⟨*proof*⟩

lemma *derive_singleton*: $P \vdash [a] \Rightarrow u \longleftrightarrow (\exists A.\ (A,u) \in P \wedge a = Nt\ A)$
⟨*proof*⟩

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*⟩

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*⟩

lemma *derives_Cons_decomp*:
$P \vdash s \# u \Rightarrow* v \longleftrightarrow$
$(\exists\, v2.\ v = s\#v2 \wedge P \vdash u \Rightarrow* v2) \vee$
$(\exists\, A\ w\ v1\ v2.\ v = v1\ @\ v2 \wedge s = Nt\ A \wedge (A,w) \in P \wedge P \vdash w \Rightarrow* v1 \wedge P \vdash u$
$\Rightarrow* v2)$ (**is** *?L ⟷ ?R*)
⟨*proof*⟩

lemma *deriven_Suc_decomp_left*:
$P \vdash u \Rightarrow(Suc\ n)\ v \longleftrightarrow (\exists\, p\ A\ u2\ w\ v1\ v2\ n1\ n2.$
$u = p\ @\ Nt\ A \# u2 \wedge v = p\ @\ v1\ @\ v2 \wedge n = n1 + n2 \wedge$
$(A,w) \in P \wedge P \vdash w \Rightarrow(n1)\ v1 \wedge$
$P \vdash u2 \Rightarrow(n2)\ v2)$ (**is** *?l ⟷ ?r*)
⟨*proof*⟩

lemma *derives_NilD*: $P \vdash w \Rightarrow* [] \Longrightarrow s \in set\ w \Longrightarrow P \vdash [s] \Rightarrow* []$
⟨*proof*⟩

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*⟩

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*⟩

lemma *derives_set_subset*:
$P \vdash u \Rightarrow* v \Longrightarrow set\ v \subseteq set\ u \cup Syms\ P$
⟨*proof*⟩

lemma *derives_nts_syms_subset*:
$P \vdash u \Rightarrow* v \Longrightarrow nts\_syms\ v \subseteq nts\_syms\ u \cup Nts\ P$
⟨*proof*⟩

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: $(A,\alpha) \in P \Longrightarrow P \vdash [Nt\ A] \Rightarrow bu\ \alpha$ |
bu_embed: $[\![P \vdash \alpha \Rightarrow bu\ \alpha_1 \mathbin{@} \alpha_2 \mathbin{@} \alpha_3;\ P \vdash \alpha_2 \Rightarrow bu\ \beta]\!] \Longrightarrow P \vdash \alpha \Rightarrow bu\ \alpha_1 \mathbin{@} \beta \mathbin{@} \alpha_3$

lemma *derives_if_bu*: $P \vdash \alpha \Rightarrow bu\ \beta \Longrightarrow P \vdash \alpha \Rightarrow * \beta$
⟨*proof*⟩

lemma *derives_bu_if*: $P \vdash \alpha \Rightarrow * \beta \Longrightarrow P \vdash \alpha \Rightarrow bu\ \beta$
⟨*proof*⟩

lemma *derives_bu_iff*: $P \vdash \alpha \Rightarrow bu\ \beta \longleftrightarrow P \vdash \alpha \Rightarrow * \beta$
⟨*proof*⟩

1.2.4 Leftmost/Rightmost Derivations

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

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

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

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

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

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

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

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

lemma *derivel_iff*: $R \vdash u \Rightarrow l\ v \longleftrightarrow$
$(\exists\ (A,w) \in R.\ \exists\, u1\ u2.\ u = map\ Tm\ u1 \mathbin{@} Nt\ A \#\ u2 \wedge v = map\ Tm\ u1 \mathbin{@} w \mathbin{@} u2)$
⟨*proof*⟩

lemma *derivel_from_empty*[*simp*]:
$P \vdash [] \Rightarrow l\ w \longleftrightarrow \mathit{False}$ ⟨*proof*⟩

lemma *deriveln_from_empty*[*simp*]:
$P \vdash [] \Rightarrow l(n)\ w \longleftrightarrow n = 0 \wedge w = []$
⟨*proof*⟩

lemma *derivels_from_empty*[*simp*]:
$\mathcal{G} \vdash [] \Rightarrow l*\ w \longleftrightarrow w = []$
⟨*proof*⟩

lemma *Un_derivel*: $R \cup S \vdash y \Rightarrow l\ z \longleftrightarrow R \vdash y \Rightarrow l\ z \vee S \vdash y \Rightarrow l\ z$
⟨*proof*⟩

lemma *derivel_append*:
$P \vdash u \Rightarrow l\ v \Longrightarrow P \vdash u\ @\ w \Rightarrow l\ v\ @\ w$
⟨*proof*⟩

lemma *deriveln_append*:
$P \vdash u \Rightarrow l(n)\ v \Longrightarrow P \vdash u\ @\ w \Rightarrow l(n)\ v\ @\ w$
⟨*proof*⟩

lemma *derivels_append*:
$P \vdash u \Rightarrow l*\ v \Longrightarrow P \vdash u\ @\ w \Rightarrow l*\ v\ @\ w$
⟨*proof*⟩

lemma *derivels1_append*:
$P \vdash u \Rightarrow l+\ v \Longrightarrow P \vdash u\ @\ w \Rightarrow l+\ v\ @\ w$
⟨*proof*⟩

lemma *derivel_Tm_Cons*:
$P \vdash \mathit{Tm}\ a\ \#\ u \Rightarrow l\ v \longleftrightarrow (\exists\, w.\ v = \mathit{Tm}\ a\ \#\ w \wedge P \vdash u \Rightarrow l\ w)$
⟨*proof*⟩

lemma *deriveln_Tm_Cons*:
$P \vdash \mathit{Tm}\ a\ \#\ u \Rightarrow l(n)\ v \longleftrightarrow (\exists\, w.\ v = \mathit{Tm}\ a\ \#\ w \wedge P \vdash u \Rightarrow l(n)\ w)$
⟨*proof*⟩

lemma *derivels_Tm_Cons*:
$P \vdash \mathit{Tm}\ a\ \#\ u \Rightarrow l*\ v \longleftrightarrow (\exists\, w.\ v = \mathit{Tm}\ a\ \#\ w \wedge P \vdash u \Rightarrow l*\ w)$
⟨*proof*⟩

lemma *derivel_Nt_Cons*:
$P \vdash \mathit{Nt}\ A\ \#\ u \Rightarrow l\ v \longleftrightarrow (\exists\, w.\ (A,w) \in P \wedge v = w\ @\ u)$
⟨*proof*⟩

lemma *derivels1_Nt_Cons*:
$P \vdash \mathit{Nt}\ A\ \#\ u \Rightarrow l+\ v \longleftrightarrow (\exists\, w.\ (A,w) \in P \wedge P \vdash w\ @\ u \Rightarrow l*\ v)$
⟨*proof*⟩

lemma *derivels_Nt_Cons*:
$P \vdash Nt\ A\ \#\ u \Rightarrow l*\ v \longleftrightarrow v = Nt\ A\ \#\ u \vee (\exists\, w.\ (A,w) \in P \wedge P \vdash w\ @\ u \Rightarrow l*\ v)$
⟨*proof*⟩

lemma *deriveln_Nt_Cons*:
$P \vdash Nt\ A\ \#\ u \Rightarrow l(n)\ v \longleftrightarrow$ (
case n *of* $0 \Rightarrow v = Nt\ A\ \#\ u$
$\mid Suc\ m \Rightarrow \exists\, w.\ (A,w) \in P \wedge P \vdash w\ @\ u \Rightarrow l(m)\ v$)
⟨*proof*⟩

lemma *derivel_map_Tm_append*:
$P \vdash map\ Tm\ w\ @\ u \Rightarrow l\ v \longleftrightarrow (\exists\, x.\ v = map\ Tm\ w\ @\ x \wedge P \vdash u \Rightarrow l\ x)$
⟨*proof*⟩

lemma *deriveln_map_Tm_append*:
$P \vdash map\ Tm\ w\ @\ u \Rightarrow l(n)\ v \longleftrightarrow (\exists\, x.\ v = map\ Tm\ w\ @\ x \wedge P \vdash u \Rightarrow l(n)\ x)$
⟨*proof*⟩

lemma *derivels_map_Tm_append*:
$P \vdash map\ Tm\ w\ @\ u \Rightarrow l*\ v \longleftrightarrow (\exists\, x.\ v = map\ Tm\ w\ @\ x \wedge P \vdash u \Rightarrow l*\ x)$
⟨*proof*⟩

lemma *derivel_not_elim_Tm*:
assumes $P \vdash xs \Rightarrow l\ map\ Nt\ w$
shows $\exists\, v.\ xs = map\ Nt\ v$
⟨*proof*⟩

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

lemma *decomp_derivel_map_Nts*:
assumes $P \vdash map\ Nt\ Xs \Rightarrow l\ map\ Nt\ Zs$
shows $\exists\, X\ Xs'\ Ys.\ Xs = X\ \#\ Xs' \wedge P \vdash [Nt\ X] \Rightarrow l\ map\ Nt\ Ys \wedge Zs = Ys\ @\ Xs'$
⟨*proof*⟩

lemma *derivel_imp_derive*: $P \vdash u \Rightarrow l\ v \Longrightarrow P \vdash u \Rightarrow v$
⟨*proof*⟩

lemma *deriveln_imp_deriven*:
$P \vdash u \Rightarrow l(n)\ v \Longrightarrow P \vdash u \Rightarrow (n)\ v$
⟨*proof*⟩

lemma *derivels_imp_derives*:
$P \vdash u \Rightarrow l*\ v \Longrightarrow P \vdash u \Rightarrow *\ v$

⟨*proof*⟩

lemma *deriveln_iff_deriven*:
P ⊢ *u* ⇒*l*(*n*) *map Tm v* ⟷ *P* ⊢ *u* ⇒(*n*) *map Tm v*
(**is** *?l* ⟷ *?r*)
⟨*proof*⟩

lemma *derivels_iff_derives*: *P* ⊢ *u* ⇒*l*∗ *map Tm v* ⟷ *P* ⊢ *u* ⇒∗ *map Tm v*
⟨*proof*⟩

lemma *deriver_iff*: *R* ⊢ *u* ⇒*r v* ⟷
(∃ (*A*,*w*) ∈ *R*. ∃ *u1 u2*. *u* = *u1* @ *Nt A* # *map Tm u2* ∧ *v* = *u1* @ *w* @ *map Tm u2*)
⟨*proof*⟩

lemma *deriver_imp_derive*: *R* ⊢ *u* ⇒*r v* ⟹ *R* ⊢ *u* ⇒ *v*
⟨*proof*⟩

lemma *derivern_imp_deriven*: *R* ⊢ *u* ⇒*r*(*n*) *v* ⟹ *R* ⊢ *u* ⇒(*n*) *v*
⟨*proof*⟩

lemma *derivers_imp_derives*: *R* ⊢ *u* ⇒*r*∗ *v* ⟹ *R* ⊢ *u* ⇒∗ *v*
⟨*proof*⟩

lemma *deriver_iff_rev_derivel*:
P ⊢ *u* ⇒*r v* ⟷ *map_prod id rev* ‘ *P* ⊢ *rev u* ⇒*l rev v* (**is** *?l* ⟷ *?r*)
⟨*proof*⟩

lemma *rev_deriver_iff_derivel*:
map_prod id rev ‘ *P* ⊢ *u* ⇒*r v* ⟷ *P* ⊢ *rev u* ⇒*l rev v*
⟨*proof*⟩

lemma *derivern_iff_rev_deriveln*:
P ⊢ *u* ⇒*r*(*n*) *v* ⟷ *map_prod id rev* ‘ *P* ⊢ *rev u* ⇒*l*(*n*) *rev v*
⟨*proof*⟩

lemma *rev_derivern_iff_deriveln*:
map_prod id rev ‘ *P* ⊢ *u* ⇒*r*(*n*) *v* ⟷ *P* ⊢ *rev u* ⇒*l*(*n*) *rev v*
⟨*proof*⟩

lemma *derivers_iff_rev_derivels*:
P ⊢ *u* ⇒*r*∗ *v* ⟷ *map_prod id rev* ‘ *P* ⊢ *rev u* ⇒*l*∗ *rev v*
⟨*proof*⟩

lemma *rev_derivers_iff_derivels*:
map_prod id rev ‘ *P* ⊢ *u* ⇒*r*∗ *v* ⟷ *P* ⊢ *rev u* ⇒*l*∗ *rev v*
⟨*proof*⟩

lemma *rev_derive*:

map_prod id rev ‘ P ⊢ u ⇒ v ⟷ P ⊢ rev u ⇒ rev v
⟨*proof*⟩

lemma *rev_deriven*:
map_prod id rev ‘ P ⊢ u ⇒(n) v ⟷ P ⊢ rev u ⇒(n) rev v
⟨*proof*⟩

lemma *rev_derives*:
map_prod id rev ‘ P ⊢ u ⇒∗ v ⟷ P ⊢ rev u ⇒∗ rev v
⟨*proof*⟩

lemma *derivern_iff_deriven*: *P ⊢ u ⇒r(n) map Tm v ⟷ P ⊢ u ⇒(n) map Tm v*
⟨*proof*⟩

lemma *derivers_iff_derives*: *P ⊢ u ⇒r∗ map Tm v ⟷ P ⊢ u ⇒∗ map Tm v*
⟨*proof*⟩

lemma *deriver_append_map_Tm*:
P ⊢ u @ map Tm w ⇒r v ⟷ (∃ x. v = x @ map Tm w ∧ P ⊢ u ⇒r x)
⟨*proof*⟩

lemma *derivern_append_map_Tm*:
P ⊢ u @ map Tm w ⇒r(n) v ⟷ (∃ x. v = x @ map Tm w ∧ P ⊢ u ⇒r(n) x)
⟨*proof*⟩

lemma *deriver_snoc_Nt*:
P ⊢ u @ [Nt A] ⇒r v ⟷ (∃ w. (A,w) ∈ P ∧ v = u @ w)
⟨*proof*⟩

lemma *deriver_singleton*:
P ⊢ [Nt A] ⇒r v ⟷ (A,v) ∈ P
⟨*proof*⟩

lemma *derivers1_snoc_Nt*:
P ⊢ u @ [Nt A] ⇒r+ v ⟷ (∃ w. (A,w) ∈ P ∧ P ⊢ u @ w ⇒r∗ v)
⟨*proof*⟩

lemma *derivers_snoc_Nt*:
P ⊢ u @ [Nt A] ⇒r∗ v ⟷ v = u @ [Nt A] ∨ (∃ w. (A,w) ∈ P ∧ P ⊢ u @ w ⇒r∗ v)
⟨*proof*⟩

lemma *derivern_snoc_Tm*:
P ⊢ u @ [Tm a] ⇒r(n) v ⟷ (∃ w. v = w @ [Tm a] ∧ P ⊢ u ⇒r(n) w)
⟨*proof*⟩

lemma *derivern_snoc_Nt*:
P ⊢ u @ [Nt A] ⇒r(n) v ⟷ (

case n of 0 ⇒ v = u @ [Nt A]
| Suc m ⇒ ∃ w. (A,w) ∈ P ∧ P ⊢ u @ w ⇒r(m) v)
⟨*proof*⟩

lemma *derivern_singleton*:
P ⊢ [Nt A] ⇒r(n) v ⟷ (
case n of 0 ⇒ v = [Nt A]
| Suc m ⇒ ∃ w. (A,w) ∈ P ∧ P ⊢ w ⇒r(m) v)
⟨*proof*⟩

1.3 Substitution in Lists

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

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

Alternative definition, but apparently no simpler to use: *substs y ys xs = concat (map (λx. 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'*
⟨*proof*⟩

lemma *substs_skip*: *y ∉ set xs ⟹ substs y ys xs = xs*
⟨*proof*⟩

lemma *susbstsNT_map_Tm*[*simp*]: *substsNt A α (map Tm w) = map Tm w*
⟨*proof*⟩

lemma *substs_len*: *length (substs y [y'] xs) = length xs*
⟨*proof*⟩

lemma *substs_rev*: *y' ∉ set xs ⟹ substs y' [y] (substs y [y'] xs) = xs*
⟨*proof*⟩

lemma *substs_der*:
(B,v) ∈ P ⟹ P ⊢ u ⇒∗ substs (Nt B) v u
⟨*proof*⟩

1.4 Epsilon-Freeness

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*

⟨*proof*⟩

lemma *Eps_free_Nil*: *Eps_free R* ⟹ (*A*,[]) ∉ *R*
⟨*proof*⟩

lemma *Eps_freeE_Cons*: *Eps_free R* ⟹ (*A*,*w*) ∈ *R* ⟹ ∃ *a u*. *w* = *a*#*u*
⟨*proof*⟩

lemma *Eps_free_derives_Nil*:
assumes *R*: *Eps_free R* **shows** *R* ⊢ *l* ⇒∗ [] ⟷ *l* = [] (**is** *?l* ⟷ *?r*)
⟨*proof*⟩

lemma *Eps_free_derivels_Nil*: *Eps_free R* ⟹ *R* ⊢ *l* ⇒*l*∗ [] ⟷ *l* = []
⟨*proof*⟩

lemma *Eps_free_deriveln_Nil*: *Eps_free R* ⟹ *R* ⊢ *l* ⇒*l*(*n*) [] ⟹ *l* = []
⟨*proof*⟩

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*⟩

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* $\Longrightarrow$ *P* $\vdash$ [*root t*] $\Rightarrow*$ *fringe t*
⟨*proof*⟩

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*) $\Longrightarrow$
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*) $\Longrightarrow$
fringes (*subst_pts t′ i ts*) =
take i (*fringes ts*) @ *fringe t′* @ *drop* (*Suc i*) (*fringes ts*)
⟨*proof*⟩

lemma *root_subst_pt*: ⟦ *i* < *length*(*fringe t*); *fringe t* ! *i* = *Nt A*; *root t′* = *Nt A* ⟧
$\Longrightarrow$ *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* ⟧
$\Longrightarrow$ *map root* (*subst_pts t′ i ts*) = *map root ts*
⟨*proof*⟩

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* ⟧
$\Longrightarrow$ *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* ⟧
$\Longrightarrow$ ∀ *t′* ∈ *set*(*subst_pts t′ i ts*). *parse_tree P t′*
⟨*proof*⟩

lemma *parse_tree_if_derives*: *P* $\vdash$ [*Nt A*] $\Rightarrow*$ *w* $\Longrightarrow$ ∃ *t*. *parse_tree P t* ∧ *fringe t* = *w* ∧ *root t* = *Nt A*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *Nt_notin_rename_syms_if_notin_range*:
x ∉ *range f* ⟹ *Nt x* ∉ *set* (*rename_syms f w*)
⟨*proof*⟩

lemma *in_Nts_rename_Prods*: *B* ∈ *Nts* (*rename_Prods f P*) = (∃ *A* ∈ *Nts P*. *f A* = *B*)
⟨*proof*⟩

lemma *rename_preserves_deriven*:
P ⊢ *α* ⇒(*n*) *β* ⟹ *rename_Prods f P* ⊢ *rename_syms f α* ⇒(*n*) *rename_syms f β*
⟨*proof*⟩

lemma *rename_preserves_derives*:
P ⊢ *α* ⇒∗ *β* ⟹ *rename_Prods f P* ⊢ *rename_syms f α* ⇒∗ *rename_syms f β*
⟨*proof*⟩

lemma *rename_preserves_derivel*:
assumes *P* ⊢ *α* ⇒*l β*
shows *rename_Prods f P* ⊢ *rename_syms f α* ⇒*l rename_syms f β*
⟨*proof*⟩

lemma *rename_preserves_deriveln*:
P ⊢ *α* ⇒*l*(*n*) *β* ⟹ *rename_Prods f P* ⊢ *rename_syms f α* ⇒*l*(*n*) *rename_syms f β*
⟨*proof*⟩

lemma *rename_preserves_derivels*:
P ⊢ *α* ⇒*l*∗ *β* ⟹ *rename_Prods f P* ⊢ *rename_syms f α* ⇒*l*∗ *rename_syms f β*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *Lang_rename_Prods*:
assumes *inj_on f* (*Nts P* ∪ {*S*})
shows *Lang* (*rename_Prods f P*) (*f S*) = *Lang P S*
⟨*proof*⟩

lemma *derives_preserves_renaming*:
assumes *rename_Prods f P* ⊢ *rename_syms f u* ⇒∗ *fv*
shows ∃ *v*. *fv* = *rename_syms f v*
⟨*proof*⟩

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 $Rhs\_Nts\ P \cap Lhss\ P' = \{\}$
and $P \cup P' \vdash u \Rightarrow l\ v$
and $nts\_syms\ u \cap Lhss\ P' = \{\}$
shows $P \vdash u \Rightarrow l\ v \wedge nts\_syms\ v \cap Lhss\ P' = \{\}$
⟨*proof*⟩

lemma *derive_disj_Un_if*:
assumes $Rhs\_Nts\ P \cap Lhss\ P' = \{\}$
and $P \cup P' \vdash u \Rightarrow v$
and $nts\_syms\ u \cap Lhss\ P' = \{\}$
shows $P \vdash u \Rightarrow v \wedge nts\_syms\ v \cap Lhss\ P' = \{\}$
⟨*proof*⟩

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*⟩

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*⟩

lemma *derivel_disj_Un_iff*:
assumes $Rhs\_Nts\ P \cap Lhss\ P' = \{\}$
and $nts\_syms\ u \cap Lhss\ P' = \{\}$
shows $P \cup P' \vdash u \Rightarrow l\ v \longleftrightarrow P \vdash u \Rightarrow l\ v$
⟨*proof*⟩

lemma *derive_disj_Un_iff*:
assumes $Rhs\_Nts\ P \cap Lhss\ P' = \{\}$
and $nts\_syms\ u \cap Lhss\ P' = \{\}$
shows $P \cup P' \vdash u \Rightarrow v \longleftrightarrow P \vdash u \Rightarrow v$
⟨*proof*⟩

lemma *deriveln_disj_Un_iff*:
assumes $Rhs\_Nts\ P \cap Lhss\ P' = \{\}$
and $nts\_syms\ u \cap Lhss\ P' = \{\}$
shows $P \cup P' \vdash u \Rightarrow l(n)\ v \longleftrightarrow P \vdash u \Rightarrow l(n)\ v$
⟨*proof*⟩

lemma *deriven_disj_Un_iff*:
assumes $Rhs\_Nts\ P \cap Lhss\ P' = \{\}$
and $nts\_syms\ u \cap Lhss\ P' = \{\}$
shows $P \cup P' \vdash u \Rightarrow (n)\ v \longleftrightarrow P \vdash u \Rightarrow (n)\ v$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *Lang_disj_Un1*:
assumes *Rhs_Nts P* ∩ *Lhss P′* = {}
and *S* ∉ *Lhss P′*
shows *Lang P S* = *Lang* (*P* ∪ *P′*) *S*
⟨*proof*⟩

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*⟩

4.2 Disjoint Union including start fork

lemma *derive_from_isolated_fork*:
⟦ *A* ∉ *Lhss P*; {(*A*,*α1*),(*A*,*α2*)} ∪ *P* ⊢ [*Nt A*] ⇒ *β* ⟧ ⟹ *β* = *α1* ∨ *β* = *α2*
⟨*proof*⟩

lemma *derives_fork_if_derives1*:
assumes *P* ⊢ [*Nt B1*] ⇒∗ *map Tm w*
shows {(*A*,[*Nt B1*]), (*A*,[*Nt B2*])} ∪ *P* ⊢ [*Nt A*] ⇒∗ *map Tm w* (**is** *?P* ⊢ _ ⇒∗ _)
⟨*proof*⟩

lemma *derives_disj_if_derives_fork*:
assumes *A* ∉ *Nts P* ∪ {*B1*,*B2*}
and {(*A*,[*Nt B1*]), (*A*,[*Nt B2*])} ∪ *P* ⊢ [*Nt A*] ⇒∗ *map Tm w* (**is** *?P* ⊢ _ ⇒∗ _)
shows *P* ⊢ [*Nt B1*] ⇒∗ *map Tm w* ∨ *P* ⊢ [*Nt B2*] ⇒∗ *map Tm w*
⟨*proof*⟩

lemma *Lang_distrib_eq_Un_Lang2*:
assumes *A* ∉ *Nts P* ∪ {*B1*,*B2*}
shows *Lang* ({(*A*,[*Nt B1*]),(*A*,[*Nt B2*])} ∪ *P*) *A* = (*Lang P B1* ∪ *Lang P B2*)
(**is** *Lang ?P* _ = _ **is** *?L1* = *?L2*)
⟨*proof*⟩

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*⟩

end

5 Context-Free Languages

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

5.1 Auxiliary: *lfp* as Kleene Fixpoint

definition *omega_chain* :: (*nat* ⇒ ('*a*::*complete_lattice*)) ⇒ *bool* **where**
omega_chain C = (∀ *i*. *C i* ≤ *C*(*Suc i*))

definition *omega_cont* :: (('*a*::*complete_lattice*) ⇒ ('*b*::*complete_lattice*)) ⇒ *bool*
where
omega_cont f = (∀ *C*. *omega_chain C* ⟶ *f*(*SUP n*. *C n*) = (*SUP n*. *f*(*C n*)))

lemma *omega_chain_mono*: *omega_chain C* ⟹ *i* ≤ *j* ⟹ *C i* ≤ *C j*
⟨*proof*⟩

lemma *mono_if_omega_cont*: **fixes** *f* :: ('*a*::*complete_lattice*) ⇒ ('*b*::*complete_lattice*)
assumes *omega_cont f* **shows** *mono f*
⟨*proof*⟩

lemma *omega_chain_iterates*: **fixes** *f* :: ('*a*::*complete_lattice*) ⇒ '*a*
assumes *mono f* **shows** *omega_chain*(λ*n*. (*f*^^*n*) *bot*)
⟨*proof*⟩

theorem *Kleene_lfp*:
assumes *omega_cont f* **shows** *lfp f* = (*SUP n*. (*f*^^*n*) *bot*) (**is** _ = *?U*)
⟨*proof*⟩

5.2 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.

lemma *arb_inj_on_finite_infinite*: *finite*(*A* :: '*a set*) ⟹ ∃ *f* :: '*a* ⇒ '*b*::*infinite*. *inj_on f A*

⟨*proof*⟩

lemma *CFL_change_Nt_type*: **assumes** *CFL TYPE*('*t1*::*infinite*) *L* **shows** *CFL TYPE*('*t2*::*infinite*) *L*
⟨*proof*⟩

For hiding the infinite type of nonterminals:

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

5.3 Closure Properties

lemma *CFL_Un_closed*:
assumes *CFL TYPE*('*n1*) *L1 CFL TYPE*('*n2*) *L2*
shows *CFL TYPE*((*'n1*+*'n2*)*option*) (*L1* ∪ *L2*)
⟨*proof*⟩

lemma *CFL_concat_closed*:
assumes *CFL TYPE*('*n1*) *L1* **and** *CFL TYPE*('*n2*) *L2*
shows *CFL TYPE*((*'n1* + *'n2*) *option*) (*L1* @@ *L2*)
⟨*proof*⟩

5.4 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*)
⟨*proof*⟩

lemma *foldr_map_mono*: *F* ≤ *G* ⟹ *foldr* (@@) (*map F xs*) *Ls* ⊆ *foldr* (@@) (*map G xs*) *Ls*
⟨*proof*⟩

lemma *inst_sym_mono*: $F \leq G \Longrightarrow inst\_sym\ F\ s \subseteq inst\_sym\ G\ s$
⟨*proof*⟩

lemma *foldr_conc_map_inst_sym*:
assumes *omega_chain L*
shows *foldr* (@@) (*map* (λ*s*. $\bigcup i$. *inst_sym* (*L i*) *s*) *xs*) *Ls* = ($\bigcup i$. *foldr* (@@) (*map* (*inst_sym* (*L i*)) *xs*) *Ls*)
⟨*proof*⟩

lemma *omega_cont_Lang_lfp*: *omega_cont* (*subst_lang P*)
⟨*proof*⟩

theorem *Lang_lfp_SUP*: *Lang_lfp P* = (*SUP n*. ((*subst_lang P*)^^*n*) (λ*A*. {}))
⟨*proof*⟩

5.5 *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* ⊆ *R' A*) ⟹ *w* ∈ *inst_syms R α* ⟹ *w* ∈ *inst_syms R' α*
⟨*proof*⟩

lemma *omega_cont_Lang_lfp_iterates*: *omega_chain* (λ*n*. ((*subst_lang P*)^^*n*) (λ*A*. {}))
⟨*proof*⟩

lemma *in_subst_langD_inst_syms*: *w* ∈ *subst_lang P L A* ⟹ ∃ *α*. (*A*,*α*)∈*P* ∧ *w* ∈ *inst_syms L α*
⟨*proof*⟩

lemma *foldr_conc_conc*: *foldr* (@@) *xs* {[]} @@ *A* = *foldr* (@@) *xs A*
⟨*proof*⟩

lemma *derives_if_inst_syms*:
w ∈ *inst_syms* (λ*A*. {*w*. *P* ⊢ [*Nt A*] ⇒∗ *map Tm w*}) *α* ⟹ *P* ⊢ *α* ⇒∗ *map Tm w*
⟨*proof*⟩

lemma *derives_if_in_subst_lang*: *w* ∈ ((*subst_lang P*)^^*n*) (λ*A*. {}) *A* ⟹ *P* ⊢ [*Nt A*] ⇒∗ *map Tm w*
⟨*proof*⟩

lemma *derives_if_Lang_lfp*: *w* ∈ *Lang_lfp P A* ⟹ *P* ⊢ [*Nt A*] ⇒∗ *map Tm w*
⟨*proof*⟩

lemma *Lang_lfp_subset_Lang*: *Lang_lfp P A* ⊆ *Lang P A*

⟨*proof*⟩

The other direction:

lemma *inst_syms_decomp*:
⟦ ∀ *i* < *length ws*. *ws* ! *i* ∈ *inst_sym L* (*α* ! *i*); *length α* = *length ws* ⟧
⟹ *concat ws* ∈ *inst_syms L α*
⟨*proof*⟩

lemma *Lang_lfp_if_derives_aux*: *P* ⊢ [*Nt A*] ⇒(*n*) *map Tm w* ⟹ *w* ∈ ((*subst_lang P*)^^*n*) (λ*A*. {}) *A*
⟨*proof*⟩

lemma *Lang_lfp_if_derives*: *P* ⊢ [*Nt A*] ⇒∗ *map Tm w* ⟹ *w* ∈ *Lang_lfp P A*
⟨*proof*⟩

theorem *Lang_lfp_eq_Lang*: *Lang_lfp P A* = *Lang P A*
⟨*proof*⟩

end

6 Elimination of Unit Productions

theory *Unit_Elimination*
imports *Context_Free_Grammar*
begin

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

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

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

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

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

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

lemma *Unit_free_if_unit_elim_rel*: *unit_elim_rel ps ps'* $\Longrightarrow$ *Unit_free* (*set ps'*)

⟨*proof*⟩

lemma *unit_elim_rel_Eps_free*:
  assumes *Eps_free* (*set ps*) **and** *unit_elim_rel ps ps'*
  shows *Eps_free* (*set ps'*)
  ⟨*proof*⟩

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

lemma *unit_prods_uprods*: *set* (*uprods ps*) = *unit_prods ps*
⟨*proof*⟩

lemma *finiteunit_prods*: *finite* (*unit_prods ps*)
⟨*proof*⟩

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

lemma *finite_unit_rtc*:
  assumes *finite ps*
  shows *finite* (*unit_rtc ps*)
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *finite_nPSlambda*:
  assumes *finite Ps*
  shows *finite* (*nPSlambda Ps d*)
⟨*proof*⟩

lemma *finite_new_prods*: *finite* (*new_prods ps*)
⟨*proof*⟩

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

lemma *unit_elim_rel_exists*: $\forall\, ps.\ \exists\, ps'.\ unit\_elim\_rel\ ps\ ps'$
⟨*proof*⟩

definition *unit_elim* **where**
$unit\_elim\ ps = (SOME\ ps'.\ unit\_elim\_rel\ ps\ ps')$

lemma *unit_elim_rel_unit_elim*: $unit\_elim\_rel\ ps\ (unit\_elim\ ps)$
⟨*proof*⟩

lemma *inNonUnitProds*:
$p \in unit\_rm\ ps \Longrightarrow p \in set\ ps$
⟨*proof*⟩

lemma *psubDeriv*:
assumes $ps \vdash u \Rightarrow v$
and $\forall\, p \in ps.\ p \in ps'$
shows $ps' \vdash u \Rightarrow v$
⟨*proof*⟩

lemma *psubRtcDeriv*:
assumes $ps \vdash u \Rightarrow* v$
and $\forall\, p \in ps.\ p \in ps'$
shows $ps' \vdash u \Rightarrow* v$
⟨*proof*⟩

lemma *unit_prods_deriv*:
assumes $unit\_prods\ ps \vdash u \Rightarrow* v$
shows $set\ ps \vdash u \Rightarrow* v$
⟨*proof*⟩

lemma *unit_elim_rel_r3*:
assumes $unit\_elim\_rel\ ps\ ps'$ **and** $set\ ps' \vdash u \Rightarrow v$
shows $set\ ps \vdash u \Rightarrow* v$
⟨*proof*⟩

lemma *unit_elim_rel_r4*:
assumes $set\ ps' \vdash u \Rightarrow* v$
and $unit\_elim\_rel\ ps\ ps'$
shows $set\ ps \vdash u \Rightarrow* v$
⟨*proof*⟩

lemma *deriv_unit_rtc*:
assumes $set\ ps \vdash [Nt\ A] \Rightarrow [Nt\ B]$
shows $(A, B) \in unit\_rtc\ (unit\_prods\ ps)$
⟨*proof*⟩

lemma *unit_elim_rel_r12*:
assumes *unit_elim_rel ps ps'* $(A, \alpha) \in$ *set ps'*
shows $(A, \alpha) \notin$ *unit_prods ps*
⟨*proof*⟩

lemma *unit_elim_rel_r14*:
assumes *unit_elim_rel ps ps'*
and *set ps* $\vdash$ [*Nt A*] $\Rightarrow$ [*Nt B*] *set ps'* $\vdash$ [*Nt B*] $\Rightarrow$ *v*
shows *set ps'* $\vdash$ [*Nt A*] $\Rightarrow$ *v*
⟨*proof*⟩

lemma *unit_elim_rel_r20_aux*:
assumes *set ps* $\vdash$ *l* @ [*Nt A*] @ *r* $\Rightarrow*$ *map Tm v*
shows $\exists \alpha$. *set ps* $\vdash$ *l* @ [*Nt A*] @ *r* $\Rightarrow$ *l* @ α @ *r* $\wedge$ *set ps* $\vdash$ *l* @ α @ *r* $\Rightarrow*$ *map Tm v* $\wedge$ $(A, \alpha) \in$ *set ps*
⟨*proof*⟩

lemma *unit_elim_rel_r20*:
assumes *set ps* $\vdash$ *u* $\Rightarrow*$ *map Tm v unit_elim_rel ps ps'*
shows *set ps'* $\vdash$ *u* $\Rightarrow*$ *map Tm v*
⟨*proof*⟩

theorem *unit_elim_rel_lang_eq*: *unit_elim_rel ps ps'* $\Longrightarrow$ *lang ps' S* = *lang ps S*
⟨*proof*⟩

corollary *lang_unit_elim*: *lang* (*unit_elim ps*) *A* = *lang ps A*
⟨*proof*⟩

end

7 Elimination of Epsilon Productions

theory *Epsilon_Elimination*
imports *Context_Free_Grammar*
begin

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

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

lemma *nullables_if*:
assumes *set ps* $\vdash$ *u* $\Rightarrow*$ *v*
and *u*=[*a*] *nullables ps v*
shows *nullables ps u*

⟨*proof*⟩

lemma *nullable_if*: *set ps* ⊢ [*a*] ⇒∗ [] ⟹ *nullable ps a*
⟨*proof*⟩

lemma *nullable_aux*: ∀ *s*∈*set gamma*. *nullable ps s* ∧ *set ps* ⊢ [*s*] ⇒∗ [] ⟹ *set ps* ⊢ *gamma* ⇒∗ []
⟨*proof*⟩

lemma *if_nullable*: *nullable ps a* ⟹ *set ps* ⊢ [*a*] ⇒∗ []
⟨*proof*⟩

corollary *nullable_iff*: *nullable ps a* ⟷ *set ps* ⊢ [*a*] ⇒∗ []
⟨*proof*⟩

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

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

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

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

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

lemma *eps_elim_fun_eq*: *eps_elim ps* = ⋃((*eps_elim_fun ps*) ‘ *set ps*)
⟨*proof*⟩

lemma *finite_eps_elim*: *finite* (*eps_elim ps*)
⟨*proof*⟩

lemma *eps_elim_rel_exists*: ∀ *ps*. ∃ *ps*'. *eps_elim_rel ps ps*'
⟨*proof*⟩

lemma *eps_closure_nullable*: [] ∈ *set* (*eps_closure ps w*) ⟹ *nullables ps w*
⟨*proof*⟩

lemma *eps_elim_rel_1*: *r*' ∈ *set* (*eps_closure ps r*) ⟹ *set ps* ⊢ *r* ⇒∗ *r*'
⟨*proof*⟩

lemma *eps_elim_rel_r2*:
assumes *set ps'* $\vdash$ *u* $\Rightarrow$ *v* **and** *eps_elim_rel ps ps'*
shows *set ps* $\vdash$ *u* $\Rightarrow*$ *v*
⟨*proof*⟩

lemma *eps_elim_rel_r3*:
assumes *set ps'* $\vdash$ *u* $\Rightarrow*$ *v* **and** *eps_elim_rel ps ps'*
shows *set ps* $\vdash$ *u* $\Rightarrow*$ *v*
⟨*proof*⟩

lemma *eps_elim_rel_r5*: *r* $\in$ *set* (*eps_closure ps r*)
⟨*proof*⟩

lemma *eps_elim_rel_r4*:
assumes (*l*,*r*) $\in$ *set ps*
and *eps_elim_rel ps ps'*
and (*r'* $\neq$ [])
and *r'* $\in$ *set* (*eps_closure ps r*)
shows (*l*,*r'*) $\in$ *set ps'*
⟨*proof*⟩

lemma *eps_elim_rel_r7*:
assumes *eps_elim_rel ps ps'*
and *set ps* $\vdash$ [*Nt A*] $\Rightarrow$ *v*
and *v'* $\in$ *set* (*eps_closure ps v*) $\wedge$ (*v'* $\neq$ [])
shows *set ps'* $\vdash$ [*Nt A*] $\Rightarrow$ *v'*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *eps_elim_rel_r15*:

assumes *set ps* ⊢ [*Nt S*] ⇒∗ *u*
and *eps_elim_rel ps ps′*
and *v* ∈ *set* (*eps_closure ps u*) ∧ (*v* ≠ [])
shows *set ps′* ⊢ [*Nt S*] ⇒∗ *v*
⟨*proof*⟩

theorem *eps_elim_rel_eq_if_noe*:
assumes *eps_elim_rel ps ps′*
and [] ∉ *lang ps S*
shows *lang ps S* = *lang ps′ S*
⟨*proof*⟩

lemma *noe_lang_eps_elim_rel_aux*:
assumes *ps* ⊢ [*Nt S*] ⇒∗ *w w* = []
shows ∃ *A*. *ps* ⊢ [*Nt S*] ⇒∗ [*Nt A*] ∧ (*A*, *w*) ∈ *ps*
⟨*proof*⟩

lemma *noe_lang_eps_elim_rel*: *eps_elim_rel ps ps′* ⟹ [] ∉ *lang ps′ S*
⟨*proof*⟩

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

end

8 Conversion to Chomsky Normal Form

theory *Chomsky_Normal_Form*
imports *Unit_Elimination Epsilon_Elimination*
begin

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

lemma *Nts_correct*: *A* ∉ *Nts P* ⟹ (∄ *S α*. (*S*, *α*) ∈ *P* ∧ (*Nt A* ∈ {*Nt S*} ∪ *set α*))
⟨*proof*⟩

definition *uniformize* :: '*n*::*infinite* ⇒ '*t* ⇒ '*n* ⇒ ('*n*,'*t*)*prods* ⇒ ('*n*,'*t*) *prods* ⇒ *bool* **where**
uniformize A t S ps ps′ ≡ (
∃ *l r p s*. (*l*,*r*) ∈ *set ps* ∧ (*r* = *p*@[*Tm t*]@*s*)
∧ (*p* ≠ [] ∨ *s* ≠ []) ∧ *A* = *fresh*(*nts ps* ∪ {*S*})
∧ *ps′* = ((*removeAll* (*l*,*r*) *ps*) @ [(*A*,[*Tm t*]), (*l*, *p*@[*Nt A*]@*s*)]))

lemma *uniformize_Eps_free*:
assumes *Eps_free* (*set ps*)
and *uniformize A t S ps ps'*
shows *Eps_free* (*set ps'*)
⟨*proof*⟩

lemma *uniformize_Unit_free*:
assumes *Unit_free* (*set ps*)
and *uniformize A t S ps ps'*
shows *Unit_free* (*set ps'*)
⟨*proof*⟩

definition *prodTms* :: ('*n*,'*t*) *prod* ⇒ *nat* **where**
prodTms p ≡ (*if length* (*snd p*) ≤ *1 then 0 else length* (*filter* (*isTm*) (*snd p*)))

definition *prodNts* :: ('*n*,'*t*) *prod* ⇒ *nat* **where**
prodNts p ≡ (*if length* (*snd p*) ≤ *2 then 0 else length* (*filter* (*isNt*) (*snd p*)))

fun *badTmsCount* :: ('*n*,'*t*) *prods* ⇒ *nat* **where**
badTmsCount ps = *sum_list*(*map prodTms ps*)

lemma *badTmsCountSet*: (∀ *p* ∈ *set ps*. *prodTms p* = *0*) ⟷ *badTmsCount ps* = *0*
⟨*proof*⟩

fun *badNtsCount* :: ('*n*,'*t*) *prods* ⇒ *nat* **where**
badNtsCount ps = *sum_list*(*map prodNts ps*)

lemma *badNtsCountSet*: (∀ *p* ∈ *set ps*. *prodNts p* = *0*) ⟷ *badNtsCount ps* = *0*
⟨*proof*⟩

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

lemma *uniform_badTmsCount*:
uniform (*set ps*) ⟷ *badTmsCount ps* = *0*
⟨*proof*⟩

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

lemma *binary_badNtsCount*:
assumes *uniform* (*set ps*) *badNtsCount ps* = *0*
shows *binary* (*set ps*)
⟨*proof*⟩

lemma *count_bin_un*: (*binary* (*set ps*) ∧ *uniform* (*set ps*)) ⟷ (*badTmsCount ps* = *0* ∧ *badNtsCount ps* = *0*)

⟨*proof*⟩

definition *binarizeNt* :: *'n*::*infinite* ⇒ *'n* ⇒ *'n* ⇒ *'n* ⇒ (*'n*,*'t*)*prods* ⇒ (*'n*,*'t*)*prods*
⇒ *bool* **where**
binarizeNt A B_1 B_2 *S ps ps'* ≡ (
∃ *l r p s*. (*l*,*r*) ∈ *set ps* ∧ (*r* = *p*@[*Nt* B_1,*Nt* B_2]@*s*)
∧ (*p* ≠ [] ∨ *s* ≠ []) ∧ (*A* = *fresh*(*nts ps* ∪ {*S*}))
∧ *ps'* = ((*removeAll* (*l*,*r*) *ps*) @ [(*A*, [*Nt* B_1,*Nt* B_2]), (*l*, *p*@[*Nt A*]@*s*)]))

lemma *binarizeNt_Eps_free*:
assumes *Eps_free* (*set ps*)
and *binarizeNt A* B_1 B_2 *S ps ps'*
shows *Eps_free* (*set ps'*)
⟨*proof*⟩

lemma *binarizeNt_Unit_free*:
assumes *Unit_free* (*set ps*)
and *binarizeNt A* B_1 B_2 *S ps ps'*
shows *Unit_free* (*set ps'*)
⟨*proof*⟩

lemma *fresh_nts_single*: *fresh*(*nts ps* ∪ {*S*}) ∉ *nts ps* ∪ {*S*}
⟨*proof*⟩

lemma *binarizeNt_aux1*:
assumes *binarizeNt A* B_1 B_2 *S ps ps'*
shows *A* ≠ B_1 ∧ *A* ≠ B_2
⟨*proof*⟩

lemma *derives_sub*:
assumes *P* ⊢ [*Nt A*] ⇒ *u* **and** *P* ⊢ *xs* ⇒ *p* @ [*Nt A*] @ *s*
shows *P* ⊢ *xs* ⇒∗ *p* @ *u* @ *s*
⟨*proof*⟩

lemma *cnf_r1Tm*:
assumes *uniformize A t S ps ps'*
and *set ps* ⊢ *lhs* ⇒ *rhs*
shows *set ps'* ⊢ *lhs* ⇒∗ *rhs*
⟨*proof*⟩

lemma *cnf_r1Nt*:
assumes *binarizeNt A* B_1 B_2 *S ps ps'*
and *set ps* ⊢ *lhs* ⇒ *rhs*
shows *set ps'* ⊢ *lhs* ⇒∗ *rhs*
⟨*proof*⟩

lemma *slemma1_1*:
assumes *uniformize A t S ps ps'*

and $(A, \alpha) \in set\ ps'$
shows $\alpha = [Tm\ t]$
⟨*proof*⟩

lemma *slemma1_1Nt*:
assumes *binarizeNt A* B_1 B_2 *S ps ps'*
and $(A, \alpha) \in set\ ps'$
shows $\alpha = [Nt\ B_1, Nt\ B_2]$
⟨*proof*⟩

lemma *slemma4_1*:
assumes $Nt\ A \notin set\ rhs$
shows $\forall \alpha.\ rhs = substsNt\ A\ \alpha\ rhs$
⟨*proof*⟩

lemma *slemma4_3_1*:
assumes $lhs = A$
shows $\alpha = substsNt\ A\ \alpha\ [Nt\ lhs]$
⟨*proof*⟩

lemma *slemma4_4*:
assumes *uniformize A t S ps ps'*
and $(l,r) \in set\ ps$
shows $Nt\ A \notin set\ r$
⟨*proof*⟩

lemma *slemma4_4Nt*:
assumes *binarizeNt A* B_1 B_2 *S ps ps'*
and $(l,r) \in set\ ps$
shows $(Nt\ A) \notin set\ r$
⟨*proof*⟩

lemma *lemma1*:
assumes *uniformize A t S ps ps'*
and $set\ ps' \vdash lhs \Rightarrow rhs$
shows $substsNt\ A\ [Tm\ t]\ lhs = substsNt\ A\ [Tm\ t]\ rhs$
$\vee\ set\ ps \vdash substsNt\ A\ [Tm\ t]\ lhs \Rightarrow substsNt\ A\ [Tm\ t]\ rhs$
⟨*proof*⟩

lemma *lemma1Nt*:
assumes *binarizeNt A* B_1 B_2 *S ps ps'*
and $set\ ps' \vdash lhs \Rightarrow rhs$
shows $(substsNt\ A\ [Nt\ B_1, Nt\ B_2]\ lhs = substsNt\ A\ [Nt\ B_1, Nt\ B_2]\ rhs)$
$\vee\ ((set\ ps) \vdash (substsNt\ A\ [Nt\ B_1, Nt\ B_2]\ lhs) \Rightarrow substsNt\ A\ [Nt\ B_1, Nt\ B_2]\ rhs)$
⟨*proof*⟩

lemma *lemma3*:

assumes *set ps′ ⊢ lhs ⇒∗ rhs*
and *uniformize A t S ps ps′*
shows *set ps ⊢ substsNt A [Tm t] lhs ⇒∗ substsNt A [Tm t] rhs*
⟨*proof*⟩

lemma *lemma3Nt*:
assumes *set ps′ ⊢ lhs ⇒∗ rhs*
and *binarizeNt A B₁ B₂ S ps ps′*
shows *set ps ⊢ substsNt A [Nt B₁, Nt B₂] lhs ⇒∗ substsNt A [Nt B₁, Nt B₂] rhs*
⟨*proof*⟩

lemma *lemma4*:
assumes *uniformize A t S ps ps′*
shows *lang ps′ S ⊆ lang ps S*
⟨*proof*⟩

lemma *lemma4Nt*:
assumes *binarizeNt A B₁ B₂ S ps ps′*
shows *lang ps′ S ⊆ lang ps S*
⟨*proof*⟩

lemma *slemma5_1*:
assumes *set ps ⊢ u ⇒∗ v*
and *uniformize A t S ps ps′*
shows *set ps′ ⊢ u ⇒∗ v*
⟨*proof*⟩

lemma *slemma5_1Nt*:
assumes *set ps ⊢ u ⇒∗ v*
and *binarizeNt A B₁ B₂ S ps ps′*
shows *set ps′ ⊢ u ⇒∗ v*
⟨*proof*⟩

lemma *lemma5*:
assumes *uniformize A t S ps ps′*
shows *lang ps S ⊆ lang ps′ S*
⟨*proof*⟩

lemma *lemma5Nt*:
assumes *binarizeNt A B₁ B₂ S ps ps′*
shows *lang ps S ⊆ lang ps′ S*
⟨*proof*⟩

lemma *cnf_lemma1*: *uniformize A t S ps ps′ ⟹ lang ps S = lang ps′ S*
⟨*proof*⟩

lemma *cnf_lemma1Nt*: *binarizeNt A B₁ B₂ S ps ps′ ⟹ lang ps S = lang ps′ S*
⟨*proof*⟩

lemma *uniformizeRtc_Eps_free*:
assumes $(\lambda x\ y.\ \exists A\ t.\ uniformize\ A\ t\ S\ x\ y)^{**}\ ps\ ps'$
and *Eps_free* (*set ps*)
shows *Eps_free* (*set ps'*)
⟨*proof*⟩

lemma *binarizeNtRtc_Eps_free*:
assumes $(\lambda x\ y.\ \exists A\ t\ B_1\ B_2.\ binarizeNt\ A\ B_1\ B_2\ S\ x\ y)^{**}\ ps\ ps'$
and *Eps_free* (*set ps*)
shows *Eps_free* (*set ps'*)
⟨*proof*⟩

lemma *uniformizeRtc_Unit_free*:
assumes $(\lambda x\ y.\ \exists A\ t.\ uniformize\ A\ t\ S\ x\ y)^{**}\ ps\ ps'$
and *Unit_free* (*set ps*)
shows *Unit_free* (*set ps'*)
⟨*proof*⟩

lemma *binarizeNtRtc_Unit_free*:
assumes $(\lambda x\ y.\ \exists A\ t\ B_1\ B_2.\ binarizeNt\ A\ B_1\ B_2\ S\ x\ y)^{**}\ ps\ ps'$
and *Unit_free* (*set ps*)
shows *Unit_free* (*set ps'*)
⟨*proof*⟩

lemma *uniformize_Nts*:
assumes *uniformize A t S ps ps'* $S \in Nts$ (*set ps*)
shows $S \in Nts$ (*set ps'*)
⟨*proof*⟩

lemma *uniformizeRtc_Nts*:
assumes $(\lambda x\ y.\ \exists A\ t.\ uniformize\ A\ t\ S\ x\ y)^{**}\ ps\ ps'\ S \in Nts$ (*set ps*)
shows $S \in Nts$ (*set ps'*)
⟨*proof*⟩

theorem *cnf_lemma2*:
assumes $(\lambda x\ y.\ \exists A\ t.\ uniformize\ A\ t\ S\ x\ y)^{**}\ ps\ ps'$
shows *lang ps S* = *lang ps' S*
⟨*proof*⟩

theorem *cnf_lemma2Nt*:
assumes $(\lambda x\ y.\ \exists A\ t\ B_1\ B_2.\ binarizeNt\ A\ B_1\ B_2\ S\ x\ y)^{**}\ ps\ ps'$
shows *lang ps S* = *lang ps' S*
⟨*proof*⟩

theorem *cnf_lemma*:

assumes $(\lambda x\ y.\ \exists A\ t.\ uniformize\ A\ t\ S\ x\ y)^{**}\ ps\ ps'$
and $(\lambda x\ y.\ \exists A\ B_1\ B_2.\ binarizeNt\ A\ B_1\ B_2\ S\ x\ y)^{**}\ ps'\ ps''$
shows $lang\ ps\ S = lang\ ps''\ S$
⟨*proof*⟩

lemma *badTmsCount_append*: $badTmsCount\ (ps@ps') = badTmsCount\ ps + badTmsCount\ ps'$
⟨*proof*⟩

lemma *badNtsCount_append*: $badNtsCount\ (ps@ps') = badNtsCount\ ps + badNtsCount\ ps'$
⟨*proof*⟩

lemma *badTmsCount_removeAll*:
assumes $prodTms\ p > 0\ p \in set\ ps$
shows $badTmsCount\ (removeAll\ p\ ps) < badTmsCount\ ps$
⟨*proof*⟩

lemma *badNtsCount_removeAll*:
assumes $prodNts\ p > 0\ p \in set\ ps$
shows $badNtsCount\ (removeAll\ p\ ps) < badNtsCount\ ps$
⟨*proof*⟩

lemma *badTmsCount_removeAll2*:
assumes $prodTms\ p > 0\ p \in set\ ps\ prodTms\ p' < prodTms\ p$
shows $badTmsCount\ (removeAll\ p\ ps) + prodTms\ p' < badTmsCount\ ps$
⟨*proof*⟩

lemma *badNtsCount_removeAll2*:
assumes $prodNts\ p > 0\ p \in set\ ps\ prodNts\ p' < prodNts\ p$
shows $badNtsCount\ (removeAll\ p\ ps) + prodNts\ p' < badNtsCount\ ps$
⟨*proof*⟩

lemma *lemma6_a*:
assumes $uniformize\ A\ t\ S\ ps\ ps'$ **shows** $badTmsCount\ (ps') < badTmsCount\ ps$
⟨*proof*⟩

lemma *lemma6_b*:
assumes $binarizeNt\ A\ B_1\ B_2\ S\ ps\ ps'$ **shows** $badNtsCount\ ps' < badNtsCount\ ps$
⟨*proof*⟩

lemma *badTmsCount0_removeAll*: $badTmsCount\ ps = 0 \Longrightarrow badTmsCount\ (removeAll\ (l,r)\ ps) = 0$
⟨*proof*⟩

lemma *slemma15_a*:
assumes $binarizeNt\ A\ B_1\ B_2\ S\ ps\ ps'$

and *badTmsCount ps = 0*
shows *badTmsCount ps' = 0*
⟨*proof*⟩

lemma *lemma15_a*:
assumes *(λx y. ∃ A B₁ B₂. binarizeNt A B₁ B₂ S x y)^** ps ps'*
and *badTmsCount ps = 0*
shows *badTmsCount ps' = 0*
⟨*proof*⟩

lemma *noTms_prodTms0*:
assumes *prodTms (l,r) = 0*
shows *length r ≤ 1 ∨ (∀ a ∈ set r. isNt a)*
⟨*proof*⟩

lemma *badTmsCountNot0*:
assumes *badTmsCount ps > 0*
shows *∃ l r t. (l,r) ∈ set ps ∧ length r ≥ 2 ∧ Tm t ∈ set r*
⟨*proof*⟩

lemma *badNtsCountNot0*:
assumes *badNtsCount ps > 0*
shows *∃ l r. (l, r) ∈ set ps ∧ length r ≥ 3*
⟨*proof*⟩

lemma *list_longer2*: *length l ≥ 2 ∧ x ∈ set l ⟹ (∃ hd tl . l = hd@[x]@tl ∧ (hd ≠ [] ∨ tl ≠ []))*
⟨*proof*⟩

lemma *list_longer3*: *length l ≥ 3 ⟹ (∃ hd tl x y. l = hd@[x]@[y]@tl ∧ (hd ≠ [] ∨ tl ≠ []))*
⟨*proof*⟩

lemma *lemma8_a*: *badTmsCount ps > 0 ⟹ ∃ ps' A t. uniformize A t S ps ps'*
⟨*proof*⟩

lemma *lemma8_b*:
assumes *badTmsCount ps = 0* **and** *badNtsCount ps > 0*
shows *∃ ps' A B₁ B₂. binarizeNt A B₁ B₂ S ps ps'*
⟨*proof*⟩

lemma *uniformize_2*: *∃ ps'. (λx y. ∃ A t. uniformize A t S x y)^** ps ps' ∧ (badTmsCount ps' = 0)*
⟨*proof*⟩

lemma *binarizeNt_2*:
assumes *badTmsCount ps = 0*
shows *∃ ps'. (λx y. ∃ A B₁ B₂. binarizeNt A B₁ B₂ S x y)^** ps ps' ∧ (badNtsCount ps' = 0)*

⟨*proof*⟩

theorem *cnf_noe_nou*: **fixes** *ps* :: (*'n*::*infinite*,*'t*)*prods*
assumes *Eps_free* (*set ps*) **and** *Unit_free* (*set ps*)
shows ∃ *ps'*::(*'n*,*'t*)*prods*. *uniform* (*set ps'*) ∧ *binary* (*set ps'*) ∧ *lang ps S* = *lang ps' S* ∧ *Eps_free* (*set ps'*) ∧ *Unit_free* (*set ps'*)
⟨*proof*⟩

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*⟩

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

theorem *cnf_exists*:
fixes *ps* :: (*'n*::*infinite*,*'t*) *prods*
shows ∃ *ps'*::(*'n*,*'t*)*prods*. *CNF*(*set ps'*) ∧ *lang ps' S* = *lang ps S* − {[]}
⟨*proof*⟩

Some helpful properties:

lemma *cnf_length_derive*:
assumes *CNF P P* ⊢ [*Nt S*] ⇒∗ α
shows *length* α ≥ *1*
⟨*proof*⟩

lemma *cnf_length_derive2*:
assumes *CNF P P* ⊢ [*Nt A*, *Nt B*] ⇒∗ α
shows *length* α ≥ *2*
⟨*proof*⟩

lemma *cnf_single_derive*:
assumes *CNF P P* ⊢ [*Nt S*] ⇒∗ [*Tm t*]
shows (*S*, [*Tm t*]) ∈ *P*
⟨*proof*⟩

lemma *cnf_word*:
assumes *CNF P P* ⊢ [*Nt S*] ⇒∗ *map Tm w*
and *length w* ≥ *2*
shows ∃ *A B u v*. (*S*, [*Nt A*, *Nt B*]) ∈ *P* ∧ *P* ⊢ [*Nt A*] ⇒∗ *map Tm u* ∧ *P* ⊢ [*Nt B*] ⇒∗ *map Tm v* ∧ *u*@*v* = *w* ∧ *u* ≠ [] ∧ *v* ≠ []
⟨*proof*⟩

end

9 Pumping Lemma for Context Free Grammars

theory *Pumping_Lemma_CFG*
imports

List_Power.List_Power
Chomsky_Normal_Form
begin

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

inductive *path* :: ('*n*, '*t*) *Prods* ⇒ '*n* ⇒ '*n list* ⇒ '*t list* ⇒ *bool*
((*2*_ ⊢/ (_/ ⇒⟨_⟩/ _)) [*50*, *0*, *0*, *50*] *50*) **where**
terminal: (*A*, [*Tm a*]) ∈ *P* ⟹ *P* ⊢ *A* ⇒⟨[*A*]⟩ [*a*] |
left: ⟦(*A*, [*Nt B*, *Nt C*]) ∈ *P* ∧ (*P* ⊢ *B* ⇒⟨*p*⟩ *w*) ∧ (*P* ⊢ *C* ⇒⟨*q*⟩ *v*)⟧ ⟹ *P* ⊢ *A* ⇒⟨*A*#*p*⟩ (*w*@*v*) |
right: ⟦(*A*, [*Nt B*, *Nt C*]) ∈ *P* ∧ (*P* ⊢ *B* ⇒⟨*p*⟩ *w*) ∧ (*P* ⊢ *C* ⇒⟨*q*⟩ *v*)⟧ ⟹ *P* ⊢ *A* ⇒⟨*A*#*q*⟩ (*w*@*v*)

inductive *cnf_derives* :: ('*n*, '*t*) *Prods* ⇒ '*n* ⇒ '*t list* ⇒ *bool*
((*2*_ ⊢/ (_/ ⇒*cnf*/ _)) [*50*, *0*, *50*] *50*) **where**
step: (*A*, [*Tm a*]) ∈ *P* ⟹ *P* ⊢ *A* ⇒*cnf* [*a*]|
trans: ⟦(*A*, [*Nt B*, *Nt C*]) ∈ *P* ∧ *P* ⊢ *B* ⇒*cnf w* ∧ *P* ⊢ *C* ⇒*cnf v*⟧ ⟹ *P* ⊢ *A* ⇒*cnf* (*w*@*v*)

lemma *path_if_cnf_derives*: *P* ⊢ *S* ⇒*cnf w* ⟹ ∃ *p*. *P* ⊢ *S* ⇒⟨*p*⟩ *w*
⟨*proof*⟩

lemma *cnf_derives_if_path*: *P* ⊢ *S* ⇒⟨*p*⟩ *w* ⟹ *P* ⊢ *S* ⇒*cnf w*
⟨*proof*⟩

corollary *cnf_path*: *P* ⊢ *S* ⇒*cnf w* ⟷ (∃ *p*. *P* ⊢ *S* ⇒⟨*p*⟩ *w*)
⟨*proof*⟩

lemma *cnf_der*:
assumes *P* ⊢ [*Nt S*] ⇒∗ *map Tm w CNF P*
shows *P* ⊢ *S* ⇒*cnf w*
⟨*proof*⟩

lemma *der_cnf*:
assumes *P* ⊢ *S* ⇒*cnf w CNF P*
shows *P* ⊢ [*Nt S*] ⇒∗ *map Tm w*
⟨*proof*⟩

corollary *cnf_der_eq*: *CNF P* ⟹ (*P* ⊢ [*Nt S*] ⇒∗ *map Tm w* ⟷ *P* ⊢ *S* ⇒*cnf w*)
⟨*proof*⟩

lemma *path_if_derives*:
assumes *P* ⊢ [*Nt S*] ⇒∗ *map Tm w CNF P*
shows ∃ *p*. *P* ⊢ *S* ⇒⟨*p*⟩ *w*
⟨*proof*⟩

lemma *derives_if_path*:
assumes $P \vdash S \Rightarrow\langle p\rangle\ w$ *CNF* P
shows $P \vdash [Nt\ S] \Rightarrow* map\ Tm\ w$
⟨*proof*⟩

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: $(A, [Tm\ a]) \in P \Longrightarrow P \vdash A \Rightarrow\langle\langle[A]\rangle\rangle\ [a]$ |
nonTerminals: $[\![(A, [Nt\ B, Nt\ C]) \in P \wedge (P \vdash B \Rightarrow\langle\langle p\rangle\rangle\ w) \wedge (P \vdash C \Rightarrow\langle\langle q\rangle\rangle\ v)]\!]$
$\Longrightarrow$
$P \vdash A \Rightarrow\langle\langle A\#(if\ length\ p \geq length\ q\ then\ p\ else\ q)\rangle\rangle\ (w@v)$

lemma *path_lpath*: $P \vdash S \Rightarrow\langle p\rangle\ w \Longrightarrow \exists\, p'.\ (P \vdash S \Rightarrow\langle\langle p'\rangle\rangle\ w) \wedge length\ p' \geq length\ p$
⟨*proof*⟩

lemma *lpath_path*: $(P \vdash S \Rightarrow\langle\langle p\rangle\rangle\ w) \Longrightarrow P \vdash S \Rightarrow\langle p\rangle\ w$
⟨*proof*⟩

lemma *lpath_length*: $(P \vdash S \Rightarrow\langle\langle p\rangle\rangle\ w) \Longrightarrow length\ w \leq 2\hat{}length\ p$
⟨*proof*⟩

lemma *step_decomp*:
assumes $P \vdash A \Rightarrow\langle[A]@p\rangle\ w$ $length\ p \geq 1$
shows $\exists\, B\ C\ p'\ a\ b.\ (A, [Nt\ B, Nt\ C]) \in P \wedge w = a@b \wedge$
$((P \vdash B \Rightarrow\langle p\rangle\ a \wedge P \vdash C \Rightarrow\langle p'\rangle\ b) \vee (P \vdash B \Rightarrow\langle p'\rangle\ a \wedge P \vdash C \Rightarrow\langle p\rangle\ b))$
⟨*proof*⟩

lemma *steplp_decomp*:
assumes $P \vdash A \Rightarrow\langle\langle[A]@p\rangle\rangle\ w$ $length\ p \geq 1$
shows $\exists\, B\ C\ p'\ a\ b.\ (A, [Nt\ B, Nt\ C]) \in P \wedge w = a@b \wedge$
$((P \vdash B \Rightarrow\langle\langle p\rangle\rangle\ a \wedge P \vdash C \Rightarrow\langle\langle p'\rangle\rangle\ b \wedge length\ p \geq length\ p') \vee$
$(P \vdash B \Rightarrow\langle\langle p'\rangle\rangle\ a \wedge P \vdash C \Rightarrow\langle\langle p\rangle\rangle\ b \wedge length\ p \geq length\ p'))$
⟨*proof*⟩

lemma *path_first_step*: $P \vdash A \Rightarrow\langle p\rangle\ w \Longrightarrow \exists\, q.\ p = [A]@q$
⟨*proof*⟩

lemma *no_empty*: $P \vdash A \Rightarrow\langle p\rangle\ w \Longrightarrow length\ w > 0$
⟨*proof*⟩

lemma *substitution*:
assumes $P \vdash A \Rightarrow\langle p1@[X]@p2\rangle\ z$
shows $\exists\, v\ w\ x.\ P \vdash X \Rightarrow\langle[X]@p2\rangle\ w \wedge z = v@w@x \wedge$
$(\forall\, w'\ p'.\ P \vdash X \Rightarrow\langle[X]@p'\rangle\ w' \longrightarrow P \vdash A \Rightarrow\langle p1@[X]@p'\rangle\ v@w'@x) \wedge$
$(length\ p1 > 0 \longrightarrow length\ (v@x) > 0)$

⟨*proof*⟩

lemma *substitution_lp*:
assumes $P \vdash A \Rightarrow\!\!\langle p1@[X]@p2\rangle\!\!\rangle\ z$
shows $\exists\, v\ w\ x.\ P \vdash X \Rightarrow\!\!\langle [X]@p2\rangle\!\!\rangle\ w \wedge z = v@w@x \wedge$
$(\forall\, w'\ p'.\ P \vdash X \Rightarrow\langle [X]@p'\rangle\ w' \longrightarrow P \vdash A \Rightarrow\langle p1@[X]@p'\rangle\ v@w'@x)$
⟨*proof*⟩

lemma *path_nts*: $P \vdash S \Rightarrow\langle p\rangle\ w \Longrightarrow set\ p \subseteq Nts\ P$
⟨*proof*⟩

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 $P \vdash A \Rightarrow\langle (([A]@p2)^{\frown\frown}(Suc\ i))\ @\ [A]@p3\rangle\ v^{\frown\frown}(Suc\ i)\ @\ w\ @\ x^{\frown\frown}(Suc\ i)$
⟨*proof*⟩

lemma *inner_pumping*:
assumes *CNF P finite P*
and $m = card\ (Nts\ P)$
and $z \in Lang\ P\ S$
and $length\ z \geq 2\hat{}(m+1)$
shows $\exists\, u\ v\ w\ x\ y\ .\ z = u@v@w@x@y \wedge length\ (v@w@x) \leq 2\hat{}(m+1) \wedge length\ (v@x) \geq 1 \wedge (\forall\, i.\ u@(v^{\frown\frown}i)@w@(x^{\frown\frown}i)@y \in Lang\ P\ S)$
⟨*proof*⟩

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 $\exists\, n.\ pumping\_property\ (Lang\ P\ S)\ n$
⟨*proof*⟩

theorem *Pumping_Lemma*:
assumes *finite* $(P :: ('n::infinite, 't)Prods)$
shows $\exists\, n.\ pumping\_property\ (Lang\ P\ S)\ n$
⟨*proof*⟩

end

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

theory *AnBnCn_not_CFL*
imports *Pumping_Lemma_CFG*
begin

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

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

declare *count_list_pow_list*[*simp*]

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

lemma *c_greater_count0*:
  assumes *x@y = [a]^^n @ [b]^^n @ [c]^^n length y≥n*
  shows *count_list x c = 0*
  ⟨*proof*⟩

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

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

⟨*proof*⟩

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

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

⟨*proof*⟩

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*⟩

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*⟩

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

⟨*proof*⟩

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

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

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

⟨*proof*⟩

theorem *anbncn_not_cfl*:
assumes *finite* (*P* :: ('*n*::*infinite*,'*a*)*Prods*)
shows *Lang P S* ≠ (⋃ *n*. {[*a*]^^*n* @ [*b*]^^*n* @ [*c*]^^*n*}) (**is** ¬ *?E*)
⟨*proof*⟩

end

end

11 CFLs Are Not Closed Under Intersection

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

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

Some lemmas:

lemma *Lang_concat*:
L1 @@ *L2* = {*word*. ∃ *w1* ∈ *L1*. ∃ *w2* ∈ *L2*. *word* = *w1* @ *w2*}
⟨*proof*⟩

lemma *deriven_same_repl*:
assumes $(A,\ u' @ [Nt\ A] @ v') \in P$
shows $P \vdash u @ [Nt\ A] @ v \Rightarrow(n)\ u @ (u'^^n) @ [Nt\ A] @ (v'^^n) @ v$
⟨*proof*⟩

Now the main proof.

lemma *an_CFL*: *CFL_TYPE*('n) ($\bigcup n.\ \{[a]^^n\}$) (**is** *CFL _ ?L*)
⟨*proof*⟩

lemma *anbn_CFL*: *CFL_TYPE*('n) ($\bigcup n.\ \{[a]^^n @ [b]^^n\}$) (**is** *CFL _ ?L*)
⟨*proof*⟩

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*⟩

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

end

12 Inlining a Single Production

theory *Inlining1Prod*
imports *Context_Free_Grammar*
begin

A single production of (A,α) can be removed from *ps* by inlining (= replacing *Nt A* by α everywhere in *ps*) without changing the language if *Nt A* $\notin$ *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'*
⟨*proof*⟩

lemma *substP_skip1*: $s \notin set\ v \Longrightarrow substP\ s\ u\ ((A,v)\ \#\ ps) = (A,v)\ \#\ substP\ s\ u\ ps$
⟨*proof*⟩

lemma *substP_skip2*: $s \notin syms\ ps \Longrightarrow substP\ s\ u\ ps = ps$
⟨*proof*⟩

lemma *substP_rev*: $Nt\ B \notin syms\ ps \Longrightarrow substP\ (Nt\ B)\ [s]\ (substP\ s\ [Nt\ B]\ ps) = ps$
⟨*proof*⟩

lemma *substP_der*:
$(A,u) \in set\ (substP\ (Nt\ B)\ v\ ps) \Longrightarrow set\ ((B,v)\ \#\ ps) \vdash [Nt\ A] \Rightarrow* u$
⟨*proof*⟩

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

lemma *if_part*:
$set\ (substP\ (Nt\ B)\ v\ ps) \vdash [Nt\ A] \Rightarrow* u \Longrightarrow set\ ((B,v)\ \#\ ps) \vdash [Nt\ A] \Rightarrow* u$
⟨*proof*⟩

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 $A \neq B$
and $B \notin lhss\ ps$
and $Nt\ B \notin set\ v$
shows $set\ ((B,v)\#ps) \vdash [Nt\ A] \Rightarrow* u \Longrightarrow set\ (substP\ (Nt\ B)\ v\ ps) \vdash [Nt\ A] \Rightarrow* substsNt\ B\ v\ u$
⟨*proof*⟩

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 $set\ ((B,v)\#ps) \vdash [Nt\ A] \Rightarrow* u \Longrightarrow set\ (substP\ (Nt\ B)\ v\ ps) \vdash [Nt\ A] \Rightarrow* u$
⟨*proof*⟩

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) ⊢ [Nt A] ⇒∗ u ⟷ set ((B,v) # ps) ⊢ [Nt A] ⇒∗ u*
⟨*proof*⟩

end

13 Transforming Long Productions Into a Binary Cascade

theory *Binarize*
imports *Inlining1Prod*
begin

lemma *funpow_fix*: **fixes** *f* :: *'a* ⇒ *'a* **and** *m* :: *'a* ⇒ *nat*
assumes ⋀*x*. *m*(*f x*) < *m x* ∨ *f x* = *x*
shows *f*((*f* ^^ *m x*) *x*) = (*f* ^^ *m x*) *x*
⟨*proof*⟩

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* :: *infinite*, *'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 = fresh (nts ps') in (A,[s0, Nt B]) # (B, u) # ps)

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

fun *count* :: (*'n*::*infinite*, *'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*::*infinite*, *'t*) *prods* ⇒ (*'n*, *'t*) *prods* **where**
binarize ps = (*binarize'* ^^ (*count ps*)) *ps*

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*)
⟨*proof*⟩

lemma *count_dec′*:
assumes *binarize′ ps* ≠ *ps*
shows *count ps* > *count* (*binarize′ ps*)
⟨*proof*⟩

lemma *binarize_ffpi*:
binarize′((*binarize′* ^^ *count x*) *x*) = (*binarize′* ^^ *count x*) *x*
⟨*proof*⟩

lemma *binarize_binary1*:
assumes *ps* = *binarize1 ps′ ps*
shows (*A*,*w*) ∈ *set*(*binarize1 ps′ ps*) ⟹ *length w* ≤ *2*
⟨*proof*⟩

lemma *binarize_binary′*:
assumes *ps* = *binarize′ ps*
shows (*A*,*w*) ∈ *set*(*binarize′ ps*) ⟹ *length w* ≤ *2*
⟨*proof*⟩

lemma *binarize_binary*: (*A*,*w*) ∈ *set*(*binarize ps*) ⟹ *length w* ≤ *2*
⟨*proof*⟩

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 ps′*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *lhss_binarize1*:
lhss ps ⊆ *lhss* (*binarize1 ps′ ps*)
⟨*proof*⟩

lemma *lhss_binarize′n*:
lhss ps ⊆ *lhss* ((*binarize′*^^*n*) *ps*)

⟨*proof*⟩

lemma *binarize_der'n*:
assumes $A \in lhss\ ps$
shows $set\ ps \vdash [Nt\ A] \Rightarrow* map\ Tm\ x \longleftrightarrow set\ ((binarize'^{\frown\frown} n)\ ps) \vdash [Nt\ A] \Rightarrow* map\ Tm\ x$
⟨*proof*⟩

lemma *binarize_der*:
assumes $A \in lhss\ ps$
shows $set\ ps \vdash [Nt\ A] \Rightarrow* map\ Tm\ x \longleftrightarrow set\ (binarize\ ps) \vdash [Nt\ A] \Rightarrow* map\ Tm\ x$
⟨*proof*⟩

lemma *lang_binarize_lhss*:
assumes $A \in lhss\ ps$
shows $lang\ ps\ A = lang\ (binarize\ ps)\ A$
⟨*proof*⟩

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 $Nt\ A \in syms\ ps$
shows $Nt\ A \in syms\ (binarize1\ ps'\ ps)$
⟨*proof*⟩

lemma *binarize_lhss_nts1*:
assumes $A \notin lhss\ ps$
and $A \in nts\ ps'$
shows $A \notin lhss\ (binarize1\ ps'\ ps)$
⟨*proof*⟩

lemma *binarize_lhss_nts'n*:
assumes $A \notin lhss\ ps$
and $A \in nts\ ps$
shows $A \notin lhss\ ((binarize'^{\frown\frown} n)\ ps) \wedge A \in nts\ ((binarize'^{\frown\frown} n)\ ps)$
⟨*proof*⟩

lemma *binarize_lhss_nts*:
assumes $A \notin lhss\ ps$
and $A \in nts\ ps$
shows $A \notin lhss\ (binarize\ ps) \wedge A \in nts\ (binarize\ ps)$
⟨*proof*⟩

lemma *binarize_nts'n*:
assumes $A \in nts\ ps$
shows $A \in nts\ ((binarize'\ ^{\frown\frown}\ n)\ ps)$
⟨*proof*⟩

lemma *binarize_nts*:
assumes $A \in nts\ ps$
shows $A \in nts\ (binarize\ ps)$
⟨*proof*⟩

lemma *lang_binarize*:
assumes $A \in nts\ ps$
shows $lang\ (binarize\ ps)\ A = lang\ ps\ A$
⟨*proof*⟩

end

14 Right-Linear Grammars

theory *Right_Linear*
imports *Unit_Elimination Binarize*
begin

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

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

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

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

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

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**
$rlin2\ ps = (\forall (A,w) \in ps.\ w = [] \vee (\exists a\ B.\ w = [Tm\ a,\ Nt\ B]))$

A straightforward property:

lemma *rlin_if_rlin2*:
assumes *rlin2 ps*
shows *rlin ps*
⟨*proof*⟩

lemma *rlin_cases*:
assumes *rlin_ps*: *rlin ps*
and *elem*: $(A,w) \in ps$

shows $(\exists B.\ w = [Nt\ B]) \vee (\exists u.\ w = map\ Tm\ u \vee (\exists B.\ w = map\ Tm\ u\ @\ [Nt\ B] \wedge length\ u > 0))$
⟨*proof*⟩

14.1 From *rlin* to *rlin2*

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

fun *finalize1* :: ('*n* :: *infinite*, '*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* = *fresh*(*nts ps′*) *in* (*A*,*w* @ [*Nt B*])#(*B*,[])#*ps else* (*A*,*w*) # *finalize1 ps′ ps*)

definition *finalize′* :: ('*n*::*infinite*, '*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*)

definition *finalize* :: ('*n*::*infinite*, '*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*)
⟨*proof*⟩

lemma *countfin_dec′*:
  assumes *finalize′ ps* ≠ *ps*
  shows *countfin ps* > *countfin* (*finalize′ ps*)
  ⟨*proof*⟩

lemma *finalize_ffpi*:
  finalize′((*finalize′* ^^ *countfin x*) *x*) = (*finalize′* ^^ *countfin x*) *x*
⟨*proof*⟩

lemma *finalize_rlinnoterm1*:
assumes *rlin* (*set ps*)
and *ps* = *finalize1 ps′ ps*
shows *rlin_noterm* (*set ps*)
⟨*proof*⟩

lemma *finalize_rlin1*:
rlin (*set ps*) ⟹ *rlin* (*set* (*finalize1 ps′ ps*))
⟨*proof*⟩

lemma *finalize_rlin′*:
rlin (*set ps*) ⟹ *rlin* (*set* (*finalize′ ps*))
⟨*proof*⟩

lemma *finalize_rlin′n*:
rlin (*set ps*) ⟹ *rlin* (*set* ((*finalize′*^^*n*) *ps*))
⟨*proof*⟩

lemma *finalize_rlinnoterm′*:
assumes *rlin* (*set ps*)
and *ps* = *finalize′ ps*
shows *rlin_noterm* (*set ps*)
⟨*proof*⟩

lemma *finalize_rlinnoterm*:
rlin (*set ps*) ⟹ *rlin_noterm* (*set* (*finalize ps*))
⟨*proof*⟩

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

lemma *finalize1_cases*:
finalize1 ps′ ps = *ps* ∨ (∃ *A w ps″ B*. *set ps* = {(*A*, *w*)} ∪ *set ps″* ∧ *set* (*finalize1 ps′ ps*) = {(*A*,*w* @ [*Nt B*]),(*B*,[])} ∪ *set ps″* ∧ *Nt B* ∉ *syms ps′*)
⟨*proof*⟩

lemma *finalize_der′*:
assumes *A* ∈ *lhss ps*
shows *set ps* ⊢ [*Nt A*] ⇒∗ *map Tm x* ⟷ *set* (*finalize′ ps*) ⊢ [*Nt A*] ⇒∗ *map Tm x*
⟨*proof*⟩

lemma *lhss_finalize1*:
lhss ps ⊆ *lhss* (*finalize1 ps′ ps*)
⟨*proof*⟩

lemma *lhss_binarize′n*:
lhss ps ⊆ *lhss* ((*finalize′*^^*n*) *ps*)
⟨*proof*⟩

lemma *finalize_der'n*:
assumes $A \in lhss\ ps$
shows $set\ ps \vdash [Nt\ A] \Rightarrow* map\ Tm\ x \longleftrightarrow set\ ((finalize'^\frown n)\ ps) \vdash [Nt\ A] \Rightarrow* map\ Tm\ x$
⟨*proof*⟩

lemma *finalize_der*:
assumes $A \in lhss\ ps$
shows $set\ ps \vdash [Nt\ A] \Rightarrow* map\ Tm\ x \longleftrightarrow set\ (finalize\ ps) \vdash [Nt\ A] \Rightarrow* map\ Tm\ x$
⟨*proof*⟩

lemma *lang_finalize_lhss*:
assumes $A \in lhss\ ps$
shows $lang\ ps\ A = lang\ (finalize\ ps)\ A$
⟨*proof*⟩

lemma *finalize_syms1*:
assumes $Nt\ A \in syms\ ps$
shows $Nt\ A \in syms\ (finalize1\ ps'\ ps)$
⟨*proof*⟩

lemma *finalize_nts'n*:
assumes $A \in nts\ ps$
shows $A \in nts\ ((finalize'\ ^\frown\ n)\ ps)$
⟨*proof*⟩

lemma *finalize_nts*:
assumes $A \in nts\ ps$
shows $A \in nts\ (finalize\ ps)$
⟨*proof*⟩

lemma *finalize_lhss_nts1*:
assumes $A \notin lhss\ ps$
and $A \in nts\ ps'$
shows $A \notin lhss\ (finalize1\ ps'\ ps)$
⟨*proof*⟩

lemma *finalize_lhss_nts'n*:
assumes $A \notin lhss\ ps$
and $A \in nts\ ps$
shows $A \notin lhss\ ((finalize'^\frown n)\ ps) \wedge A \in nts\ ((finalize'^\frown n)\ ps)$
⟨*proof*⟩

lemma *finalize_lhss_nts*:
assumes $A \notin lhss\ ps$
and $A \in nts\ ps$
shows $A \notin lhss\ (finalize\ ps) \wedge A \in nts\ (finalize\ ps)$

⟨*proof*⟩

lemma *lang_finalize*:
assumes $A \in nts\ ps$
shows $lang\ (finalize\ ps)\ A = lang\ ps\ A$
⟨*proof*⟩

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 $rlin\_bin\ (set\ ps)$
⟨*proof*⟩

lemma *binarize_noterm1*:
$rlin\_noterm\ (set\ ps) \Longrightarrow rlin\_noterm\ (set\ (binarize1\ ps'\ ps))$
⟨*proof*⟩

lemma *binarize_noterm'*:
$rlin\_noterm\ (set\ ps) \Longrightarrow rlin\_noterm\ (set\ (binarize'\ ps))$
⟨*proof*⟩

lemma *binarize_noterm'n*:
$rlin\_noterm\ (set\ ps) \Longrightarrow rlin\_noterm\ (set\ ((binarize'\widehat{\ }\widehat{\ }n)\ ps))$
⟨*proof*⟩

lemma *binarize_rlinbin'*:
assumes $rlin\_noterm\ (set\ ps)$
and $ps = binarize'\ ps$
shows $rlin\_bin\ (set\ ps)$
⟨*proof*⟩

lemma *binarize_rlinbin*:
$rlin\_noterm\ (set\ ps) \Longrightarrow rlin\_bin\ (set\ (binarize\ ps))$
⟨*proof*⟩

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

lemma *uppr_rlin2*:
assumes *rlinbin*: $rlin\_bin\ (set\ ps')$
and *uppr_ps'*: $unit\_elim\_rel\ ps'\ ps$
shows $rlin2\ (set\ ps)$
⟨*proof*⟩

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*::*infinite*,'*t*) *prods* ⇒ ('*n*,'*t*)*prods* **where**
rlin2_of_rlin ps = *unit_elim* (*binarize* (*finalize ps*))

theorem *rlin_to_rlin2*:
assumes *rlin* (*set ps*)
shows *rlin2* (*set* (*rlin2_of_rlin ps*))
⟨*proof*⟩

lemma *lang_rlin2_of_rlin*:
A ∈ *Nts* (*set ps*) ⟹ *lang* (*rlin2_of_rlin ps*) *A* = *lang ps A*
⟨*proof*⟩

14.2 Properties of *rlin2* derivations

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

lemma *map_Tm_single_nt*:
assumes *map Tm w* @ [*Tm a*, *Nt A*] = *u1* @ [*Nt B*] @ *u2*
shows *u1* = *map Tm* (*w* @ [*a*]) ∧ *u2* = []
⟨*proof*⟩

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* = []
⟨*proof*⟩

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

lemma *rlin2_introduce_tm*:
assumes *rlin2 P*
and *P* ⊢ [*Nt A*] ⇒∗ *map Tm w* @ [*Tm x*, *Nt B*]
shows ∃ *C*. *P* ⊢ [*Nt A*] ⇒∗ *map Tm w* @ [*Nt C*] ∧ (*C*,[*Tm x*, *Nt B*]) ∈ *P*
⟨*proof*⟩

lemma *rlin2_nts_derive_eq*:
assumes *rlin2 P*
and *P* ⊢ [*Nt A*] ⇒∗ [*Nt B*]
shows *A* = *B*
⟨*proof*⟩

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

lemma *rlin2_tms_eps*:
assumes *rlin2 P*
and $P \vdash [Nt\ A] \Rightarrow* map\ Tm\ w$
shows $\exists B.\ P \vdash [Nt\ A] \Rightarrow* map\ Tm\ w\ @\ [Nt\ B] \wedge (B,[]) \in P$
⟨*proof*⟩

end

15 Strongly Right-Linear Grammars as a Nondeterministic Automaton

theory *NDA_rlin2*
imports *Right_Linear*
begin

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

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

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

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

lemma *nxt_rlin2_nts*:
assumes $B{\in}nxt\_rlin2\ P\ A\ a$
shows $B{\in}Nts\ P$
⟨*proof*⟩

lemma *nxts_rlin2_set_app*:
nxts_rlin2_set P M (*x* @ *y*) = *nxts_rlin2_set P* (*nxts_rlin2_set P M x*) *y*
⟨*proof*⟩

lemma *nxt_rlin2_set_pick*:
assumes $B \in nxt\_rlin2\_set\ P\ M\ a$
shows $\exists C{\in}M.\ B \in nxt\_rlin2\_set\ P\ \{C\}\ a$
⟨*proof*⟩

lemma *nxts_rlin2_set_pick*:
assumes $B \in nxts\_rlin2\_set\ P\ M\ w$
shows $\exists C{\in}M.\ B \in nxts\_rlin2\_set\ P\ \{C\}\ w$
⟨*proof*⟩

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*⟩

lemma *nxts_trans0*:
assumes $B \in nxts\_rlin2\_set\ P\ (nxts\_rlin2\_set\ P\ \{A\}\ x)\ z$
shows $B \in nxts\_rlin2\_set\ P\ \{A\}\ (x@z)$
⟨*proof*⟩

lemma *nxt_mono*:
assumes $A \subseteq B$
shows $nxt\_rlin2\_set\ P\ A\ a \subseteq nxt\_rlin2\_set\ P\ B\ a$
⟨*proof*⟩

lemma *nxts_mono*:
assumes $A \subseteq B$
shows $nxts\_rlin2\_set\ P\ A\ w \subseteq nxts\_rlin2\_set\ P\ B\ w$
⟨*proof*⟩

lemma *nxts_trans1*:
assumes $M \subseteq nxts\_rlin2\_set\ P\ \{A\}\ x$
and $B \in nxts\_rlin2\_set\ P\ M\ z$
shows $B \in nxts\_rlin2\_set\ P\ \{A\}\ (x@z)$
⟨*proof*⟩

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

lemma *nxts_to_mult_derive*:
$B \in nxts\_rlin2\_set\ P\ M\ w \Longrightarrow (\exists\, A{\in}M.\ P \vdash [Nt\ A] \Rightarrow* map\ Tm\ w\ @\ [Nt\ B])$
⟨*proof*⟩

lemma *mult_derive_to_nxts*:
assumes *rlin2 P*
shows $A{\in}M \Longrightarrow P \vdash [Nt\ A] \Rightarrow* map\ Tm\ w\ @\ [Nt\ B] \Longrightarrow B \in nxts\_rlin2\_set\ P\ M\ w$
⟨*proof*⟩

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

definition $accepted\ P\ A\ w = (\exists\, Z \in nxts\_rlin2\_set\ P\ \{A\}\ w.\ (Z,[]) \in P)$

theorem *accepted_if_Lang*:
assumes *rlin2 P*
and $w \in Lang\ P\ A$
shows *accepted P A w*
⟨*proof*⟩

theorem *Lang_if_accepted*:
assumes *accepted P A w*
shows $w \in Lang\ P\ A$
⟨*proof*⟩

theorem *Lang_iff_accepted_if_rlin2*:
assumes *rlin2 P*
shows $w \in Lang\ P\ A \longleftrightarrow accepted\ P\ A\ w$
⟨*proof*⟩

end

16 Relating Strongly Right-Linear Grammars and Automata

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

16.1 From Strongly Right-Linear Grammar to NFA

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

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

interpretation *NFA_rlin2*: *nfa nfa_rlin2 P S*
⟨*proof*⟩
print_theorems

lemma *nfa_init_nfa_rlin2*: *nfa.init* (*nfa_rlin2 P S*) = *hf_of* ‘ $\{S\}$

⟨*proof*⟩

lemma *nfa_final_nfa_rlin2*: *nfa.final* (*nfa_rlin2 P S*) = *hf_of* ‘ {*A* ∈ *Nts P*. (*A*,[]) ∈ *P*}
⟨*proof*⟩

lemma *nfa_nxt_nfa_rlin2*: *nfa.nxt* (*nfa_rlin2 P S*) (*hf_of A*) *a* = *hf_of* ‘ *nxt_rlin2 P A a*
⟨*proof*⟩

lemma *nfa_epsclo_nfa_rlin2*: *M* ⊆ {*hf_of S*} ∪ *hf_of* ‘ *Nts P* ⟹ *nfa.epsclo* (*nfa_rlin2 P S*) *M* = *M*
⟨*proof*⟩

lemma *nfa_nextl_nfa_rlin2*: *M* ⊆ {*S*} ∪ *Nts P*
⟹ *nfa.nextl* (*nfa_rlin2 P S*) (*hf_of* ‘ *M*) *xs* = *hf_of* ‘ *nxts_rlin2_set P M xs*
⟨*proof*⟩

lemma *lang_pres_nfa_rlin2*: **assumes** *rlin2 P*
shows *nfa.language* (*nfa_rlin2 P S*) = *Lang P S*
⟨*proof*⟩

lemma *regular_if_rlin2*: **assumes** *rlin2 P*
shows *regular* (*Lang P S*)
⟨*proof*⟩

end

16.2 From DFA to Strongly Right-Linear Grammar

context *dfa*
begin

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

definition *Prods_dfa* :: (*hf*, *'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*)
⟨*proof*⟩

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*⟩

lemma *nxtl_to_mult_derive*:
  assumes $A \in dfa.states\ M$
    shows $Prods\_dfa \vdash [Nt\ A] \Rightarrow* map\ Tm\ w\ @\ [Nt\ (nextl\ A\ w)]$
⟨*proof*⟩

theorem *Prods_dfa_iff_dfa*:
  $q \in dfa.states\ M \Longrightarrow Prods\_dfa \vdash [Nt\ q] \Rightarrow* map\ Tm\ w \longleftrightarrow nextl\ q\ w \in dfa.final\ M$
⟨*proof*⟩

corollary *dfa_language_eq_Lang*: $dfa.language\ M = Lang\ Prods\_dfa\ (dfa.init\ M)$
⟨*proof*⟩

end

corollary *rlin2_if_regular*:
  $regular\ L \Longrightarrow \exists\, P\ S{::}hf.\ rlin2\ P \wedge L = Lang\ P\ S$
⟨*proof*⟩

end

17 Pumping Lemma for Strongly Right-Linear Grammars

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

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

lemma *not_distinct*:
  assumes $m = card\ P$
      and $m \geq 1$
      and $\forall\, i < length\ w.\ w\ !\ i \in P$
      and $length\ w \geq Suc\ m$
    shows $\exists\, xs\ ys\ zs\ y.\ w = xs\ @\ [y]\ @\ ys\ @\ [y]\ @\ zs \wedge length\ (xs\ @\ [y]\ @\ ys\ @\ [y]) \leq Suc\ m$
⟨*proof*⟩

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

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

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

17.1 Properties of *nxts_nts* and *nxts_nts0*

lemma *nxts_nts0_i0*:
∀ *e* ∈ *nxts_nts0 P A w*. *e*!*0* = *A*
⟨*proof*⟩

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*
⟨*proof*⟩

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′*
⟨*proof*⟩

lemma *nxts_nts0_len*:
∀ *e* ∈ *nxts_nts0 P A w*. *length e* = *Suc* (*length w*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *nxts_nts_elem*:
assumes *i* < *length w*
shows ∀ *e* ∈ *nxts_nts P A w*. *e* ! *i* ∈ *Nts P*
⟨*proof*⟩

lemma *nxts_nts0_elem*:
assumes *A* ∈ *Nts P*

and $i \leq$ *length* w
shows $\forall\, e \in$ *nxts_nts0* $P\ A\ w.\ e\ !\ i \in$ *Nts* P
⟨*proof*⟩

lemma *nxts_nts0_pick*:
assumes $B \in$ *nxts_rlin2_set* $P\ \{A\}\ w$
shows $\exists\, e \in$ *nxts_nts0* $P\ A\ w.$ *last* $e = B$
⟨*proof*⟩

17.2 Pumping Lemma

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

lemma *nxts_split_cycle*:
assumes *finite* P
and $A \in$ *Nts* P
and $m =$ *card* (*Nts* P)
and $B \in$ *nxts_rlin2_set* $P\ \{A\}\ w$
and *length* $w \geq m$
shows $\exists\, x\ y\ z\ C.\ w = x@y@z \wedge$ *length* $y \geq 1 \wedge$ *length* $(x@y) \leq m \wedge$
$C \in$ *nxts_rlin2_set* $P\ \{A\}\ x \wedge C \in$ *nxts_rlin2_set* $P\ \{C\}\ y \wedge B \in$ *nxts_rlin2_set* $P\ \{C\}\ z$
⟨*proof*⟩

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* $P\ \{A\}\ (x@(y^\frown i))$
⟨*proof*⟩

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* $P\ A\ (x@(y^\frown i)@z))$
⟨*proof*⟩

theorem *pumping_lemma_re_nts*:
assumes *rlin2* P

and *finite P*
and $A \in Nts\ P$
shows $\exists n.\ \forall w \in Lang\ P\ A.\ length\ w \geq n \longrightarrow$
$(\exists x\ y\ z.\ w = x@y@z \wedge length\ y \geq 1 \wedge length\ (x@y) \leq n \wedge (\forall i.\ x@(y^{\frown\frown}i)@z \in Lang\ P\ A))$
⟨*proof*⟩

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*⟩

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 $\forall n.\ \exists w \in Lang\ P\ A.\ length\ w \geq n \wedge (\forall x\ y\ z.\ w = x@y@z \wedge length\ y \geq 1 \wedge length\ (x@y) \leq n \longrightarrow (\exists i.\ x@(y^{\frown\frown}i)@z \notin Lang\ P\ A))$
shows $\neg rlin2\ P$
⟨*proof*⟩

end

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

theory *AnBn_Not_Regular*
imports *Pumping_Lemma_Regular*
begin

lemma *pow_list_set_if*: $set\ (w^{\frown\frown}k) = (if\ k{=}0\ then\ \{\}\ else\ set\ w)$
⟨*proof*⟩
lemma *in_pow_list_set*[*simp*]: $x \in set\ (ys^{\frown\frown}m) \longleftrightarrow x \in set\ ys \wedge m \neq 0$
⟨*proof*⟩
lemma *pow_list_eq_appends_iff*:
$n \geq m \Longrightarrow x^{\frown\frown}n\ @\ y = x^{\frown\frown}m\ @\ z \longleftrightarrow z = x^{\frown\frown}(n-m)\ @\ y$
⟨*proof*⟩
lemma *append_eq_append_conv_if_disj*:
$(set\ xs \cup set\ xs') \cap (set\ ys \cup set\ ys') = \{\}$
$\Longrightarrow\ xs\ @\ ys = xs'\ @\ ys' \longleftrightarrow xs = xs' \wedge ys = ys'$
⟨*proof*⟩
lemma *pow_list_eq_single_appends_iff*[*simp*]:
$[\![\ x \notin set\ ys;\ x \notin set\ zs\ ]\!] \Longrightarrow [x]^{\frown\frown}m\ @\ ys = [x]^{\frown\frown}n\ @\ zs \longleftrightarrow m = n \wedge ys = zs$
⟨*proof*⟩

The following theorem proves that the language $a^{\frown}nb^{\frown}n$ cannot be pro-

duced by a right linear production set, using the contrapositive form of the pumping lemma

theorem *not_rlin2_ab*:
  assumes $a \neq b$
    and *Lang P A* $= (\bigcup n.\ \{[a]^{\frown\frown}n\ @\ [b]^{\frown\frown}n\})$ (**is** _ $=$ *?AnBn*)
    and *finite P*
  shows $\neg$*rlin2 P*
⟨*proof*⟩

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.